FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Touch-Screen User Interface mit haptischem Feedback

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Andreas Schneider
Matrikelnummer 0525335

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: ao. Univ. Prof. Dr. Wolfgang Zagler

Wien, 05.07.2011

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Schneider Andreas
Wetzles 2, 3970 Weitra

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

_____ _____

Ort, Datum Unterschrift Verfasser

Kurzfassung

Problemstellung

Das Ziel dieser Arbeit ist es, ein universelles multimodales Userinterface zu entwerfen. Es soll möglich sein, das Gerät für verschiedene Zwecke zu verwenden. Es soll also eine Plattform entwickelt werden, die später individuell konfigurierbar ist. Zur Visualisierung soll ein Touchscreen zum Einsatz kommen. Die Größe dieses Touchscreens soll so gewählt werden, dass auch ältere Menschen gut damit arbeiten können. Konfigurierbar soll das Gerät über eine SD-Karte sein. Weiters sollen hier auch die anzuzeigenden Daten hinterlegt werden. Wichtig ist, dass der Touchscreen intuitiv bedienbar ist. Dazu soll ein neues Konzept zur Bedienung entwickelt werden, das den klassischen Doppelklick ersetzt. Insbesondere werden die Positionswahl und die Auslösung entkoppelt. Das Führen des Fingers über den Screen markiert die gewünschte Position, erst ein mechanischer Druck mit eindeutigem haptischem Feedback löst die gewählte Aktion aus. So soll das System vor allem für ältere Menschen einfacher zu bedienen sein.

Erwartetes Resultat

Das erwartete Resultat ist die hard- und softwaremäßige Entwicklung eines demonstrationsfähigen Prototyps mit den oben genannten Eigenschaften.

Methodisches Vorgehen

Der erste Schritt ist eine Recherche über verfügbare Prozessoren, Displays, Rams, etc. einschließlich eines Literaturstudiums zum Stand betreffend tangible User Interfaces. Danach soll im zweiten Schritt eine Leiterplatte entworfen werden. Im dritten Schritt sollen der Prototyp getestet und Treiber für die Hardware programmiert werden. Zuletzt soll die Funktionsweise getestet und eine Dokumentation erstellt werden.

State-of-the Art

Elektronische Geräte sind in den letzten Jahren im Haushalt unverzichtbar geworden. So ist es heute auch für ältere Menschen kaum mehr möglich, der Technik zu entkommen. Zumindest eine Regelung für die Heizung oder Lüftungssteuerung kann man in nahezu jedem Haushalt finden. Doch gerade diese sind für ältere Menschen schwierig zu bedienen. Der Grund dafür ist, dass die User Interfaces nicht an die jeweiligen Bedürfnisse angepasst werden können. Es handelt sich viel mehr um generische und sehr abstrakt gehaltene Oberflächen. Zu diesem Zweck soll das neue Gerät individuell an die betroffene Person angepasst werden können. Dadurch soll eine Vertrautheit mit bereits Bekanntem assoziiert werden können und eine einfach zu bedienende Konfigurationsschnittstelle kreiert werden.

Abstract

Presentation of the problem

The objective of this work is the design of a universal user interface. It should be possible to use this device for different applications. Therefore, an individual configurable platform should be constructed.
A touch-screen is used for visualization. The size of this screen is catered to the needs of older persons. This device should be configurable by an SD-card. In addition, all data shown should be stored there.
It is important to use an intuitive usable touch screen. For this purpose it is necessary to find a concept for the use of the screen without needing the classic double-clicks. Particularly the choice of the position and the activation should be decoupled. Touching the screen marks the favoured position, but only a mechanical pressure with a positive haptic feedback triggers the desired activity. Thus it should be possible and easy for older people to use this system.

Expected Result

The expected result is the hard- and software development of a prototype with the above-named features.

Methodical Advance

The first step is the inquiry about available processors, displays, RAMs etc. including studying descriptive literature on tangible user interfaces. Afterward a
PCB is created.
Next, the prototype gets tested and the driver unit for the hardware programmed.
In a final step the operating mode is tested and improved and finally the results are documented.

State-of-the-Art

In the last years electronic equipment got essential in every home. It is almost impossible to live in civilization without using electronic devices. That is why even older people have to learn to handle them. But most of the user interfaces are unassimilated to the needs of the people of that stage of life. Interfaces are usually designed generically and abstractly.
Because of this, the new device should be individual adjustable to the operator. Every user should be able to associate the interface with a well-known activity. Thereby an easy-to-use configuration gateway should be created.

Inhaltsverzeichnis

1 Einleitung

Seit Beginn des Informationszeitalters gewinnen EDV Systeme kontinuierlich an Bedeutung. Nicht nur im Geschäftsbereich ist die digitale Datenverarbeitung nicht mehr wegzudenken, sondern auch im privaten Bereich zählt diese längst zum Alltag. Sei es nun in der Gebäudeautomation, Heizungstechnik, Fahrkartenautomaten, etc.. Die Summe dieser Systeme führt dazu, dass ein Durchschnittsbürger/-in mehrmals täglich mit einem dieser Systeme konfrontiert ist.
Um die Automation, den Komfort oder die Funktionalität noch weiter zu erhöhen, wird in allen Bereichen an Erweiterungen und Erneuerungen gearbeitet. Der wachsende Funktionsumfang führt jedoch zu immer größeren Herausforderungen an das Mensch-Maschinen-Interface (kurz MMI oder oft auch als HMI – Human-Machine-Interface bezeichnet). Bedingt durch die wachsende Komplexität erhöht sich auch der Datendurchsatz für diese Interfaces. Daraus resultierend entstehen zwei Probleme, die es zu lösen gilt. Zum einen soll dieses Interface für alle Menschen aus verschiedensten Altersgruppen und Bildungsschichten gut bedienbar sein und zum anderen muss dieses MMI auch für den erforderlichen Datendurchsatz geeignet sein. Das Problem hierbei liegt darin, dass der Mensch es gewohnt ist, mit seiner Umwelt mit all seinen Sinnesorganen zu kommunizieren. Wenn er nun jedoch mit der „Digitalen Welt“ kommunizieren will, steht meist nur eine sehr bandbegrenzte Kommunikationsschnittstelle zur Verfügung, oder der Datenaustausch für Ein- und Ausgabe findet über unterschiedliche Medien statt. Dies führt jedoch zu einer wesentlichen Beeinträchtigung der Bedienbarkeit, da es kein direktes Feedback mehr gibt. Das klassische Beispiel hierfür wäre ein Desktop PC mit einer grafischen Benutzeroberfläche (meist als GUI - Graphical User Interface bezeichnet). Hierbei findet der Datenaustausch über zwei getrennte physikalische Schnittstellen statt. Zum einen werden die Daten grafisch auf dem Monitor dargestellt – zum anderen besteht die Möglichkeit, mit Hilfe eines Zeigergerätes Daten zu manipulieren. Jedoch stehen diese beiden Kommunikationsschnittstellen nicht direkt miteinander im Zusammenhang. In den meisten Fällen wird dieses Zeigergerät durch eine Maus realisiert. Für die Datenmanipulation kann man mit Hilfe der Maus auf die am Monitor dargestellten Objekte navigieren und diese mit Tasten bearbeiten. Jedoch sei angemerkt, dass die dreidimensionale Position der Maus nicht ident ist mit der

Position des dargestellten Objektes. Die dreidimensionale Position der Maus wird mehr oder weniger linear auf das Anzeigegerät übertragen. Dadurch entsteht eine weitere Abstraktion, die bei der Bedienung – bewusst oder unbewusst – gehandhabt werden muss.
Eine bessere Abstraktion eines MMI zwischen einem Computer und einem Anwender/-in stellt ein Touchscreen-Interface dar. Auch hier sei gleich zu Beginn angemerkt, dass es auch da Nachteile gibt. Der große Vorteil dieses Interface liegt im direkten Zusammenhang zwischen dargestelltem Symbol und Bedienposition. Soll ein Symbol, das im dreidimensionalen Raum an der Position (x,y,z) dargestellt wird, bearbeitet werden, kann es genau an dieser Position (x,y,z) ausgewählt werden. Es entfällt also das Navigieren. Da dieses Interface durch die zuvor beschriebene Funktion sehr intuitiv bedienbar ist, wird es auch in der Praxis häufig eingesetzt. Vor allem wenn es für eine sehr breite Benutzergruppe konzipiert sein soll. Wie bereits im Vorfeld erwähnt, gibt es bei diesem Interface auch einen gravierenden Nachteil. Es gibt nicht, wie bei der Bedienung mit der Maus, das klassische haptische Feedback bei der Auswahl. Dies entsteht bei der Maus durch die mechanische Betätigung einer Taste – bei einem Touchdisplay entfällt dieses. Ein weiteres Problem, das bei der Benutzung von Touchdisplays auftaucht, ist, dass die Positionswahl und die Auslösung nicht voneinander entkoppelt sind. Das bedeutet, sobald eine Position auswählt wird, wird diese zugleich auch ausgelöst. Das wiederum kann bei enger Anordnung der Symbole oder schlechter Erfassung der Touch-Position dazu führen, dass es zu häufigen Falscheingaben kommt, was wiederum zu einer schlechteren und langsameren Bedienung führt.

2 Hintergrund der Arbeit und Aufgabenstellung

2.1 *Hintergrund*

Mensch-Maschinen-Interfaces treten in unserem Alltag immer häufiger auf. Nicht nur im Leben in der Gesellschaft, sondern auch im privaten Bereich sind diese in unserem täglichen Leben integriert. Ein Problem ist, dass diese oftmals sehr generisch aufgebaut sind, um viele Anwendungsgebiete abzudecken. Das beste Beispiel hierfür ist die Heim- und Gebäudeautomation sowie Heizungs- und Lüftungssteuerung. Hier sind die Bedienelemente darauf ausgelegt, alle möglichen Funktionen eines Regelgerätes zu konfigurieren und zu bedienen. Oftmals wird – damit mit einem Bedieninterface das Auslangen gefunden wird – auf die Trennung von Bedien- und Konfigurationsinterface verzichtet. Nun sind solche Regelgeräte jedoch so konzipiert, dass sie in möglichst vielen Anlagen eingesetzt werden können. Dies führt zu dem Problem, dass das Eingabeinterface sehr generisch aufgebaut werden muss, worunter natürlich die intuitive Bedienbarkeit leidet.
Wir wollen das Ganze an folgendem Beispiel veranschaulichen. Ausgegangen wird davon, dass ein Hersteller von Lüftungsanlagen eine Regelung für seine Geräte entwerfen will. Es soll sich dabei um eine zentrale Einheit handeln, die sechs unabhängige Einzelraumlüftungen steuern kann. In jedem Raum soll sich ein Sensor befinden, sodass die Regelung Kenntnis über den Zustand der Luftqualität der einzelnen Räume besitzt. Nun will der Hersteller natürlich, dass seine Regelung in möglichst vielen verschiedenen Anlagen eingesetzt werden kann und gibt daher seinen Sensoreingängen generische Namen. (z.B. S1, S2, …,S6) Das Problem, das daraus entsteht, ist natürlich der Verlust der intuitiven Bedienbarkeit. Wenn bei einem Endkunden/-in eine solche Anlage installiert ist und man beispielsweise die Luftmenge in der Küche variieren möchte, wird man auf der Regelung keinen Eintrag „Küche“ finden können. Man muss stattdessen die Küche über den generischen Namen „S-x“ ansprechen. Besser wäre es, wenn man das Gerät so konfigurieren könnte, dass es exakt die Anlage für die es verwendet wird darstellen würde. So

könnte man die Anlage viel intuitiver bedienen und das Problem der Abstraktion würde entfallen.

2.2 *Aufgabe*

Im Nachfolgenden soll nun ein Gerät vorgestellt werden, das diese Bedürfnisse erfüllt. Dieses Gerät verfügt über einen SD- Karten Slot und soll über diesen mit Anzeigedaten versorgt werden. Dadurch soll es möglich sein, das Gerät ohne großen Aufwand auch auf sehr individuelle Bedürfnisse anzupassen. Weiters soll bei der Auswahl der Komponenten darauf geachtet werden, dass das Gerät am Ende in einem preislich akzeptablen Rahmen liegt. Dadurch, sowie durch die einfache Konfiguration, soll demonstriert werden, dass das Gerät auch für die Praxis von Interesse ist.

3 Zielsetzung

3.1 *Allgemein*

Das Ziel dieser Arbeit ist die Entwicklung eines User Interfaces, welches für verschiedene Anwendungsfälle und Zwecke adaptierbar ist. Am Ende soll ein Demonstrationsboard entstehen, welches frei über eine SD Karte konfigurierbar ist. Mit Hilfe der SD-Card soll es möglich sein, das Gerät schnell und einfach anzupassen, um die jeweiligen Bedürfnisse zu erfüllen. Eine SD-Karte soll gewählt werden, da diese am Markt sehr kostengünstig erhältlich sind, sich auch größere Datenmengen, wie z.B. Bilder darauf abspeichern lassen und kein spezielles Programmiergerät dafür benötigt wird.
Ein besonderes Augenmerk bei dieser Demonstrationsplattform soll dem Mensch-Maschinen-Interface zukommen. Es soll ein sehr intuitiv zu bedienendes Interface basierend auf einem Touchscreen zum Einsatz kommen. Die Größe und die Auflösung des Touchscreens soll so gewählt werden, dass auch ältere Menschen mit diesem Interface gut zurechtkommen. Um die Eingabe und intuitive Bedienbarkeit weiter zu steigern soll jedoch kein herkömmlicher Touchscreen verwendet werden, sondern eine Neuentwicklung bezüglich der Eingabetechnik stattfinden. Der klassische Doppelklick soll durch den Touchscreen ersetzt werden. Ziel ist es also, die Zielwahl und die Auswahl zu entkoppeln und in eigenständige Aktionen zu untergliedern. Des weiteren soll die Auswahl nicht nur wie sonst üblich visuell dargestellt werden, sondern durch ein eindeutiges haptisches Feedback bestätigt werden.

3.2 *HMI*

Konkret soll das Interface so aussehen, dass ein Führen des Fingers oder des Stiftes am Bildschirm das ausgewählte Objekt markiert und erst eine mechanische Betätigung des Bildschirms die Aktion auslöst. Dadurch soll es vereinfacht werden, Objekte anzuvisieren und vor allem älteren Menschen soll es damit einfacher fallen, sich auf dem Bildschirm zurechtzufinden und durch Menüs zu navigieren. Die Menüs und die Anzeigeseiten sollen auf der SD – Karte hinterlegt werden, um es einfacher

zu machen, das Gerät auf spezielle Einsatzgebiete oder spezielle einzelwünsche anzupassen.

3.3 *Zusätzliche Schnittstelle*

Um nicht lediglich als eigenständiges Gerät fungieren zu können, soll die Demonstrationsplattform zusätzlich über eine CAN-Bus Schnittstelle verfügen. Dadurch soll die Möglichkeit bestehen, das Gerät mit anderen Anlagen zu verbinden.

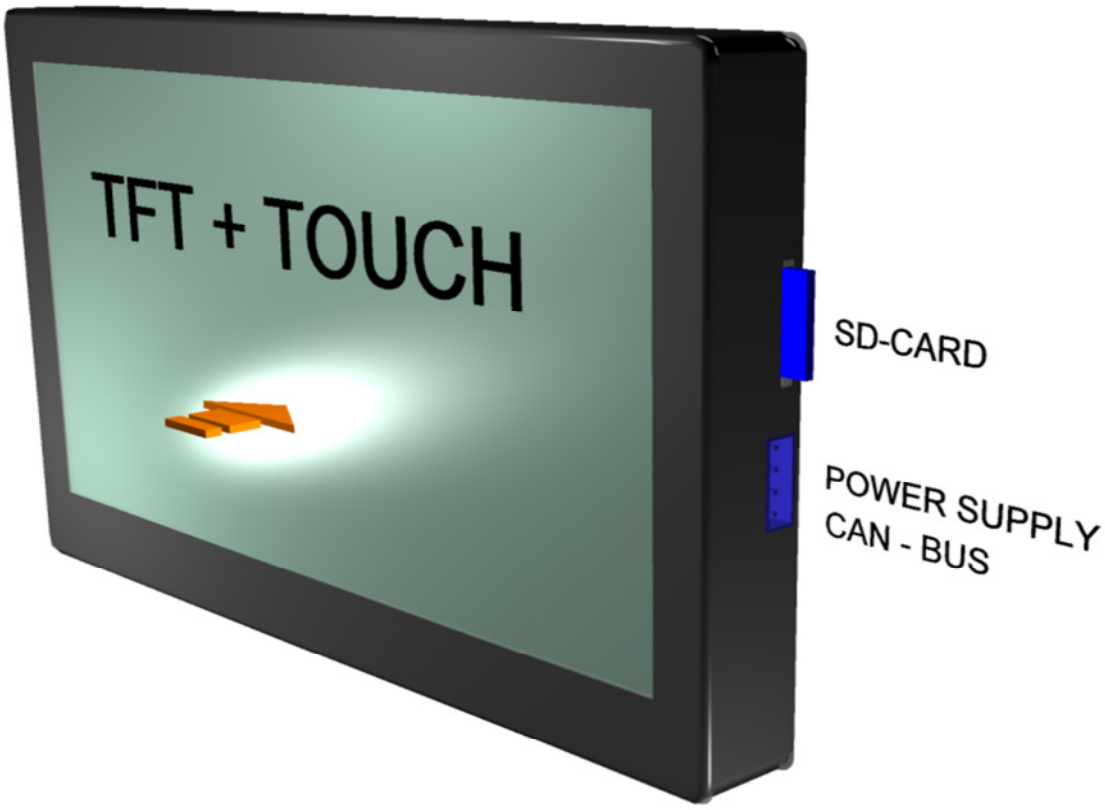

Abbildung 1: Konzept

4 Entwicklung der Hardware

4.1 *Display Evaluierung*

4.1.1 *Auswahlkriterien*

Um der Aufgabenstellung gerecht zu werden, sollte dem Display besondere Bedeutung zugewandt werden. Die Größe sollte so gewählt werden, dass auch ältere Menschen damit gut zurechtkommen. Denn weiter war klar vorgegeben, dass es sich um ein Farb/Grafik Display handeln sollte. In den meisten Anwendungen sind, wenn überhaupt, Grafik Displays (aus Kostengründen sehr kleine Displays) verbaut, die zum einen nur schwer zu lesen sind und zum anderen – wenn es sich um ein Display mit Touchfunktion handelt – nur schwer zu bedienen sind.
Genau diese Problematik ergab sich auch bei unserer Displaywahl. Da die fertige Entwicklungsplattform am Ende demonstrieren soll, dass ein frei adaptierbares Eingabeinterface von wirtschaftlichem Interesse sein kann, konnten wir die Displaygröße nicht beliebig wählen. Auch muss man für ein solches Eingabegerät später bei der Montage Platz finden können. Nichts desto trotz stand für uns die Bedienbarkeit im Vordergrund.
Ein weiteres Auswahlkriterium für uns war das Hardware Interface des Displays. Es sind Displays erhältlich, die einen integrierten Controller enthalten. Dies hat den Vorteil, dass die Displaydaten nicht ständig refresht werden müssen (Auflösung 800*480 x 16bit Farbtiefe und 50Hz Bildwiederholrate = 36,6 Mbyte/sec.), sondern nur einmalig auf das Display übertragen werden müssen. Dadurch braucht man keinen externen Grafikcontroller und die Leitungsführung auf der Printplatte (kurz PCB – printed circuit board) würde sich durch die reduzierte Anzahl an Leitungen deutlich vereinfachen. Leider sind solche Displays sehr kostspielig, wodurch wir auf diese Art von Interfaces verzichten mussten.
Auch bei Displays, die ständig refresht werden müssen, gibt es zwei häufig auftretende Typen bezüglich der Hardwareschnittstelle. Zum einen kann das Display über ein paralleles Interface verfügen, zum anderen ein LVDS Interface haben. LVDS bedeutet Low Voltage Differential Signalling. Hier werden die sonst bei einem parallelen Interface üblichen 22 Signale (6bit Rot, 6bit Grün, 6bit Blau, Clock,

DE(Data Enable), HSYNC, VSYNC) in vier Datenströme, die jeweils über zwei Leitungen übertragen werden, aufgeteilt. Durch die differenzielle Übertragung ist LVDS unempfindlicher gegenüber elektromagnetischen Störungen und die Leiterbahnführung vereinfacht sich, da die Anzahl der Leitungen reduziert sind. Weiters hat diese Reduzierung der Leitungen den Vorteil, dass das Kabel zwischen PCB und Display flexibler wird und sich die Konstruktion des Gehäuses vereinfacht. Den Nachteil einer LVDS Schnittstelle stellen wiederum die zusätzlichen Kosten für einen LVDS Transceiver dar.
Ein weiteres Unterscheidungsmerkmal von Displays liegt in der Hintergrundbeleuchtung. Hier unterscheidet man zwischen Kaltkathodenlampen (CCFL – cold cathode fluorescent lamp) und Leuchtdioden (LED – light emitting diode). CCFL Hintergrundbeleuchtungen haben zwar den Vorteil, dass sie sehr hell sind, dafür haben Sie aber eine begrenzte Lebensdauer und brauchen eine Hochspannung sowohl im laufenden Betrieb als auch vor allem beim Zünden. LEDs hingegen haben eine lange Lebensdauer (>50k Std. Betrieb) und sind vergleichsweise einfach zu regeln. Näheres dazu später.
Unter diesen Bedingungen galt es nun ein geeignetes Display zu finden.

4.1.2 *Die engere Auswahl*

Nach langer Recherche bei sämtlichen Elektronik Distributoren kamen für uns zwei Displays in die engere Auswahl. Die Eckdaten sind in der Tabelle (Tabelle 1) zusammengefasst.

Tabelle 1: Display Vergleich

Hersteller	Admatec	Tianma Micro-Electronics
Bezeichnung	NLC800T70D480CTMK29	TM104SBH01
Größe	7"	10.4"
Format	16:9	4:3
Sichtbarer Bereich	152.4 x 91.44	211.2 x 158.4
Auflösung	800 x 480	800 x 600
Farbtiefe	18 Bit	18 Bit
Hintergrundbeleuchtung	LED	LED
Interface	18bit Parallel RGB	LVDS
Touchscreen	4wire-Resistiv	4wire-Resistiv
Preis	~70€	~90€

7“ Display: Der große Vorteil dieses Displays ist das 16:9 Format, welches immer mehr Verbreitung findet. Vor allem bei der Visualisierung von Menüs kann dieses Format seine Stärke ausspielen. Zusätzlich spricht für dieses Display das 18bit Parallel Interface, das den Einsatz eines zusätzlichen LVDS Transceivers überflüssig macht. Nicht zuletzt ein weiterer Pluspunkt für dieses Display ist natürlich der Preis.

10“ Display: Hier ist der klare Vorteil die größere Darstellungsfläche, und dass das Verbindungskabel zwischen Elektronik und Display weniger Adern benötigt, da die Anzeigedaten über LVDS übertragen werden.

4.1.3 *Entscheidung*

Im Laufe des Projekts wurden beide Displays getestet. Das 10“ Modell konnte, wie erwartet, durch seine Größe und dem flexibleren Anschlusskabel punkten, jedoch

besitzt auch das 7“ Modell beinahe die selbe physikalische Auflösung. Der Preis, der Entfall des LVDS Transceivers und das stabilere Gehäuse des 7“ Displays waren am Ende der Ausschlaggeber dafür, dass wir uns für das etwas kleinere Modell entschieden haben.

4.2 *Prozessor*

4.2.1 *Anforderungen*

Die erste Anforderung an unsere Hardware ist das Display. Da wir keine bewegten Bilder (Filme) auf unserem Display darstellen wollen, sind wir der Meinung, dass wir keinen eigenen Grafik Controller benötigen, sondern versuchen sollten, dass dieser im Prozessor integriert ist. Das bedeutet aber, dass unser Prozessor einen Displaycontroller integriert haben muss, der zumindest eine Auflösung von 800x480 bei 16 Bit Farbtiefe unterstützt. (Für das 10“ Display sogar 800x600 bei 16 Bit).
Außerdem sollte der Prozessor über einen integrierten ADC verfügen, um den Touch direkt auswerten zu können (mind. zwei Prozessorports als ADC-Eingang schaltbar). Dadurch würde sich wiederum ein externer Controller einsparen lassen und sich somit der Gesamtaufwand an Komponenten und schließlich der Gesamtpreis des Gerätes minimieren.
Eine weitere Anforderung an den Controller ist ein SD-Karten Interface. Dies kann entweder durch ein Serial Peripheral Interface (SPI) erfolgen oder aber durch ein eigenes SD-Karten Interface. Die zweite Lösung hat den Vorteil, dass nicht nur eine Datenleitung wie bei SPI verwendet wird, sondern alle vier Datenleitungen der SD-Karte parallel. Dadurch lässt sich der Datendurchsatz zwischen SD-Karte und Prozessor wesentlich erhöhen.
Ein weiteres Plus für den Prozessor wäre ein integrierter CAN Controller. Da eine CAN Schnittstelle in der Aufgabenstellung gefordert ist, würde das wieder Vereinfachungen und eine Kostenersparnis mit sich bringen.

4.2.2 *Auswahl*

Unter diesen Grundvoraussetzungen suchten wir nun nach geeigneten Prozessoren. Vor allem der integrierte Displaycontroller mit der geforderten Auflösung stellte sich als der entscheidende Kriterium in unserem Systemdesign heraus. Letztendlich kamen zwei Derivate in die engere Auswahl. Zum einen der LPC2478 von NXP (früher Phillips) und zum anderen der AT91SAM9RL64 von ATMEL. Beide dieser Prozessoren haben einen integrierten LCD-Controller, welcher die geforderte Auflösung erfüllen kann. Außerdem haben beide Modelle ein Speicherkarteninterface sowie einen ADC, um die Touchfolie direkt auszuwerten. Letztendlich fiel die Entscheidung auf das Modell von NXP aus folgenden drei Gründen:
NXP hat zusätzlich noch einen integrierten CAN-Controller und macht daher den externen Einsatz eines CAN-Controllers überflüssig.
Ausschlaggebend war vor allem der zweite Grund, nämlich der integrierte Programmspeicher (FLASH), in dem der auszuführende Programmcode direkt und dauerhaft abgespeichert werden kann, welchen nur das Modell von NXP besitzt.
Der dritte Grund, der für NXP sprach, war, dass der LPC2478 in einem LQFP208 Gehäuse verfügbar ist, und nicht nur (im Vergleich zu Atmel) im BGA Gehäuse (bei LQFP sind die Pins des Prozessors außen rund um den Prozessor angeordnet, bei BGA unterhalb des Prozessors. LQFP hat also den Vorteil, dass man wesentlich leichter mit dem Oszilloskop messen und debuggen kann).

4.2.3 *Zusätzlich benötigte Hardware*

Da nun der Prozessor ausgesucht und bestimmt war, war der nächste Schritt die Evaluierung der zusätzlich benötigten Komponenten.
Da das Display eine physikalische Auflösung von 800x480 (alternativ 800x600) Bildpunkten hat und eine physikalische Farbtiefe von 18 Bit, benötigt es zur Ansteuerung (das Display wird in Wirklichkeit nur mit 16bit/2Byte angesteuert, um mit weniger Bandbreite das Auslangen zu finden/die Bildwiederholfrequenz zu steigern) einen Frambuffer von 800x480x16bit = 750 KByte (938KByte in der 10“ Variante). Zusätzlich wird, damit Objekte am Hintergrundbild dargestellt werden können, noch einmal mindestens dasselbe an RAM ein zweites Mal benötigt, um das Hintergrundbild puffern und dynamische Inhalte einblenden zu können. Daraus ergibt sich ein Speicherbedarf an Arbeitsspeicher von wenigstens 2 Mbyte. (Wir weisen hier ausdrücklich immer auf Byte hin, da bei RAM’s häufig Mbit angegeben werden.)

Da der LPC2478 lediglich über 98 kByte internes RAM verfügt, muss zusätzlich noch externes RAM an den Prozessor angeschlossen werden. Das ist auch das vom Hersteller angedachte Verfahren, um den LCD-Controller zu betreiben. Der LCD-Controller kann nämlich, nachdem er richtig konfiguriert ist, direkt per DMA (DMA – Direct Memory Access) die Anzeigedaten aus einem externen RAM laden und am Display darstellen. Für die externe RAM Erweiterung wählten wir SDRAM (Synchronous Dynamic Random Access Memory), da diese Art von RAM sehr kostengünstig und auch mit größeren Speichergrößen erhältlich ist.
Zudem besitzt der EMC (External Memory Controller) des Prozessors die Möglichkeit, externen Speicher unter anderem über einen 16bit oder 32bit breiten Bus anzukoppeln. Wir entschieden uns in unserem Design auf die 32bit breite Ankopplung, um einen höheren Datendurchsatz zu erreichen. Allerdings haben unsere Speicherbausteine nicht wirklich einen 32 Bit breiten Datenbus sondern lediglich 16bit. Um diese dennoch 32bit breit an unseren Controller anzuschließen wurden zwei dieser Bausteine parallel geschaltet. Der Grund dafür ist, dass 16bit breite Speicherbausteine weiter verbreitet sind (d.h. leichter am Markt erhältlich sind) und dadurch 2 x 16Bit breite Bausteine billiger sind als ein 32 Bit breiter Baustein bei selber Gesamtspeichergröße.
Die anderen geforderten Komponenten (SD-Karten Interface, CAN-Controller, ADC,) sind im Prozessor integriert, auf diese soll in den anschließenden Kapiteln eingegangen werden.
Das nachfolgende Blockschaltbild (Abbildung 2) soll nun die wichtigsten Komponenten unseres Systems veranschaulichen.

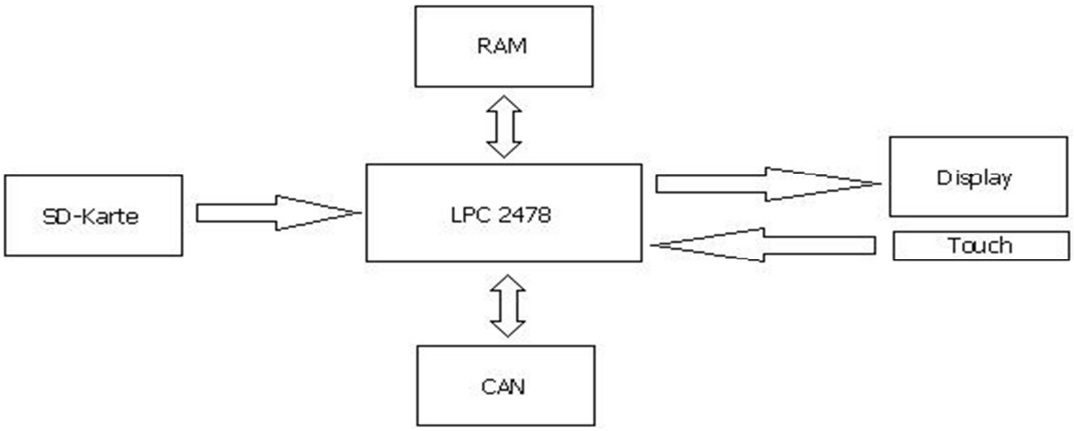

Abbildung 2: Blockschaltbild

4.3 *Spannungsversorgung*

Die Spannungsversorgung des Boards geschieht entweder über eine separate Spannungsbuchse oder aber direkt über den CAN-Bus Stecker. Der CAN-Bus Stecker auf dem Demonstrationsboard ist vierpolig ausgeführt, sodass für die Spannungsversorgung kein zusätzliches Kabel notwendig ist (CAN High, CAN Low, GND, +12V). Soll kein CAN-Bus verwendet werden kann die Spannungsversorgung über ein handelsübliches Steckernetzteil erfolgen. Damit auf die Polung keine Rücksicht genommen werden muss, befindet sich in diesem Fall am Eingang der Spannungsversorgung ein Brückengleichrichter. Je nachdem woher die Spannungsversorgung stammt, muss entweder Jumper J5 oder J6 gesetzt werden. Jumper J3 muss in jedem Fall gesetzt werden – dieser dient ausschließlich zur Inbetriebnahme des Prototypen, um zu testen, ob die Spannungsversorgung funktioniert, bevor der Rest der Schaltung mit Spannung versorgt wird. Die Spannungsversorgung der Schaltung erfolgt über einen Fixspannungsregler, da die weiteren Bauteile in dieser Schaltung alle auf 3.3V ausgelegt sind. Der hier verwendete Fixspannungsregler LM1117 von National Semiconductor im TO-252 Gehäuse ist zwar für 800mA Ausgangsstrom ausgelegt, jedoch möchten wir hier anmerken, dass bei Betrieb an 12V Eingangsspannung für die nötige Kühlung zu sorgen ist.

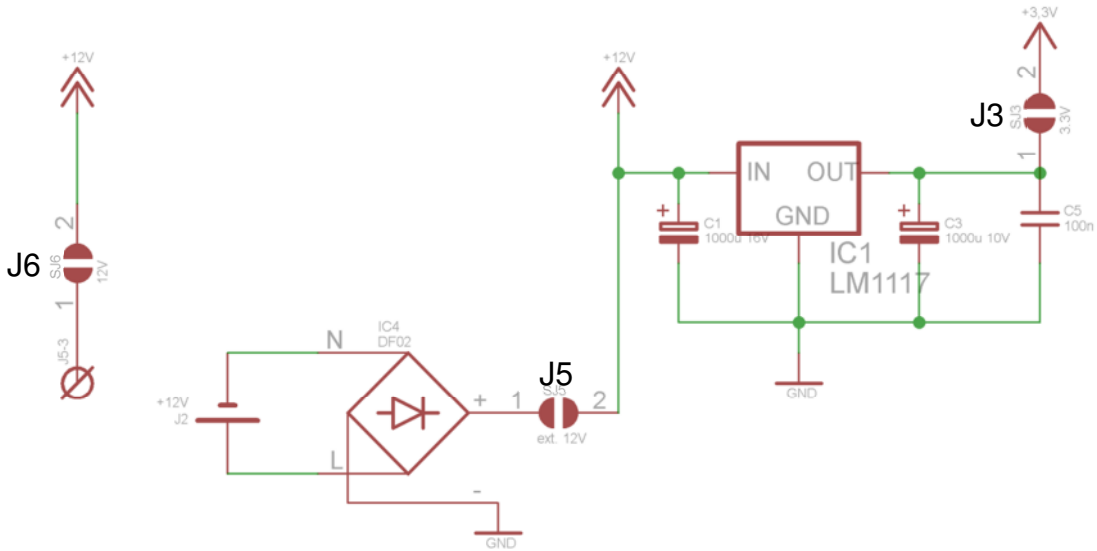

Abbildung 3: Spannungsversorgung

4.4 *Debug-Anbindung*

4.4.1 *Allgemein*

Zum Programmieren und zum Debugen des Prozessors kommt die JTAG Schnittstelle des LPC2478 zum Einsatz. Über diese ist es möglich, den Prozessor zu programmieren – also das interne Flash mit Software zu beschreiben – sowie Breakpoints zu setzen.

4.4.2 *Breakpoints*

Der Prozessor verfügt über zwei in Hardware integrierte Breakpoints. Jeder dieser Breakpoints kann als eigentlicher Breakpoint verwendet werden. Das bedeutet, sobald eine definierte Codezeile ausgeführt wird hält das System. Oder aber er wird als so genannter Watchpoint verwendet. Das bedeutet, sobald auf ein Register oder eine RAM Adresse ein definierter Wert geschrieben wird, wird die Ausführung des Programms unterbrochen. In beiden Fällen hält die Ausführung des Programmcodes und mit Hilfe eines Entwicklungsstudios können die Register des Prozessors eingesehen und verändert werden. Weiters können auch während der Programmausführung Meldungen an die Entwicklungsumbgebung ausgegeben werden, die der Programmierer/-in während der Programmausführung live sieht. Diese Debugmechanismen bringen bei der Softwareentwicklung erhebliche Erleichterungen mit sich und verkürzen so die Softwareentwicklungszeit wesentlich.

4.4.3 *Zusätzlichen Programmierfeature*

An dieser Stelle sei erwähnt, dass der Prozessor ein weiteres zusätzliches Feature zum Programmieren der Firmware besitzt. So gibt es im Befehlssatz des Prozessors die Möglichkeit, das Flash aus dem laufenden Programm heraus zu verändern. Somit lässt sich eine Updatefunktion auch ohne JTAG-Adapter implementieren. Beispielsweise kann in die Software eine Funktion integriert werden, durch die sich die Software des Prozessors über die SD-Karte selbst aktualisiert.

4.4.4 *Beschaltung*

Im nachfolgenden Schaltungsausschnitt (Abbildung 4) sehen Sie die Beschaltung des JTAG-Interfaces.

Dargestellt wird hier auch der Taktgeber des Prozessors sowie der Uhrenbaustein. Der Prozessor wird in dieser Schaltung mit einem 8MHz Quarz betrieben. Das ist aber nicht die intern verwendete Schaltfrequenz, sondern dient als Eingang für die interne PLL.
Intern wird durch die PLL ein Prozessortakt von 72MHz erreicht, was zugleich der maximalen Taktrate des Prozessors entspricht.
Für den Uhrenbaustein befindet sich zusätzlich ein 32.768 kHz Quarz auf der Platine.
Weiters sieht man im folgenden Schaltungsausschnitt noch die Beschaltung des Referenzeinganges des ADC. Der LC Tiefpass am Eingang dient zur Unterdrückung von Störungen in der Betriebsspannung. Mit Hilfe des ADC wird im Weiteren der Touchscreen ausgewertet.

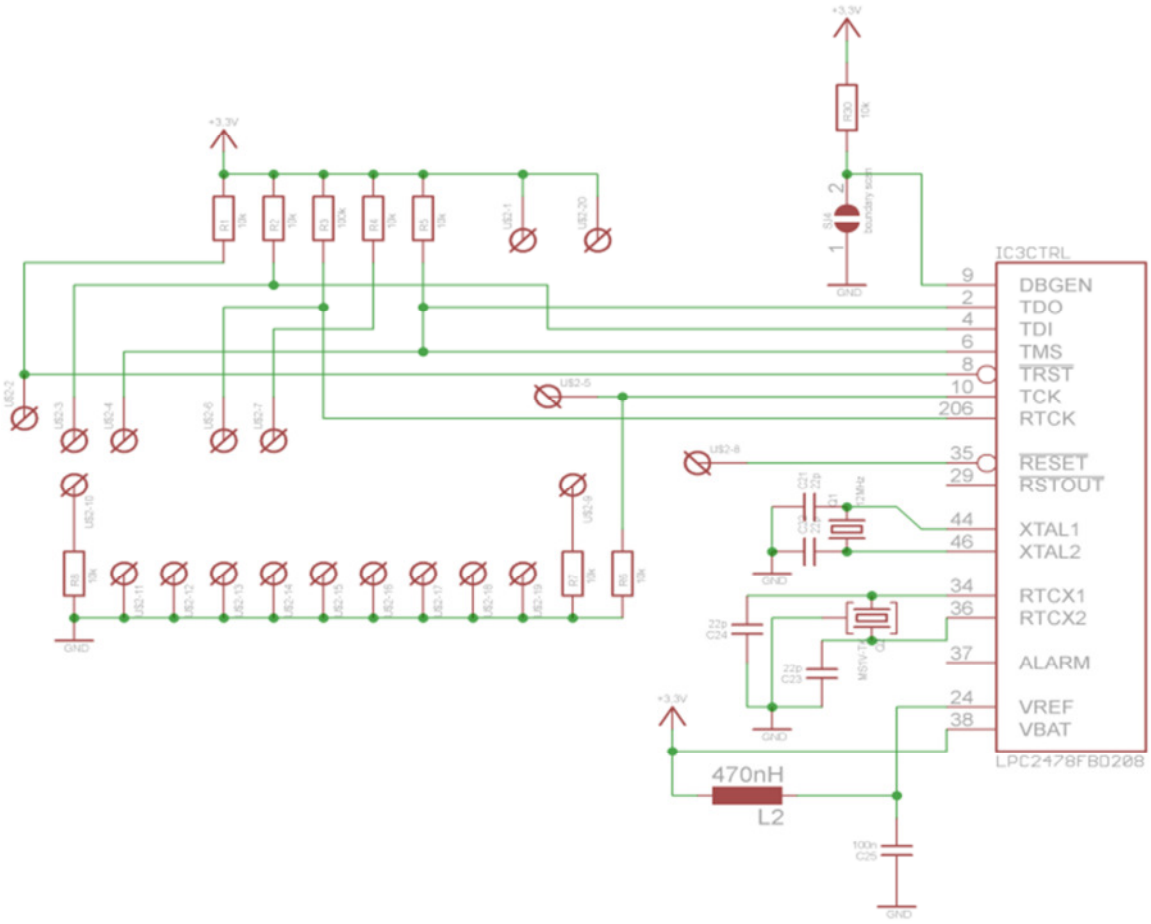

Abbildung 4: Debug Anbindung

4.5 *SD-Karten Anbindung*

4.5.1 *Allgemein*

Die SD-Karte ist an das SD/MMC Card Interface des Prozessors angebunden. Zwar könnte eine SD-Karte auch über ein Serial Peripheral Interface betrieben werden, doch wäre hier die Datenübertragungsrate nicht so hoch. Der Grund dafür ist, dass im SPI Mode nur eine Datenleitung verwendet wird – über das SD/MMC Card Interface werden jedoch alle vier Datenleitungen der SD-Karte genutzt.
Für die Datenübertragung zwischen Prozessor und SD-Karte werden die Leitungen MCIDAT[0]-MCIDAT[3], MCICMD, MCICLK benötigt, wobei alle Leitungen – bis auf MCICLK – einen Pullup-Widerstand benötigen.
Der von uns verwendete Kartenhalter stammt von Hirose (siehe Stückliste) und hat zusätzliche Kontakte für Karten- und Schreibschutzerkennung. Diese Kontakte sind ebenfalls zum Prozessor geführt.

4.5.2 *Versorgung*

Ein weiterer Pin, der für die Kommunikation mit der SD-Karte benötigt wird, ist der MCIPWR Pin. Dieser ist die Spannungsversorgung für die SD-Karte und es kann in der Software festgelegt werden, ob dieser im eingeschalteten Zustanden den High oder Low Pegel annehmen soll. Da die SD-Karte Ströme von >>10mA benötigen kann, ist es nicht möglich diesen Portpin als direkte Spannungsversorgung zu verwenden. Stattdessen wurde hier eine Transistorschaltung mittels eines FET-Transistors genommen. Zu beachten ist hier, dass der Millereffekt des Transistors absichtlich ausgenutzt wird und durch die Kapazität C44 noch zusätzlich erhöht wird. Das bewirkt, dass der Transistor langsamer schaltet. Dies hat zwar einen negativen Effekt auf die Verlustleistung des Transistors, dafür kommt es jedoch zu keinen Spannungseinbrüchen in der restlichen Schaltung.

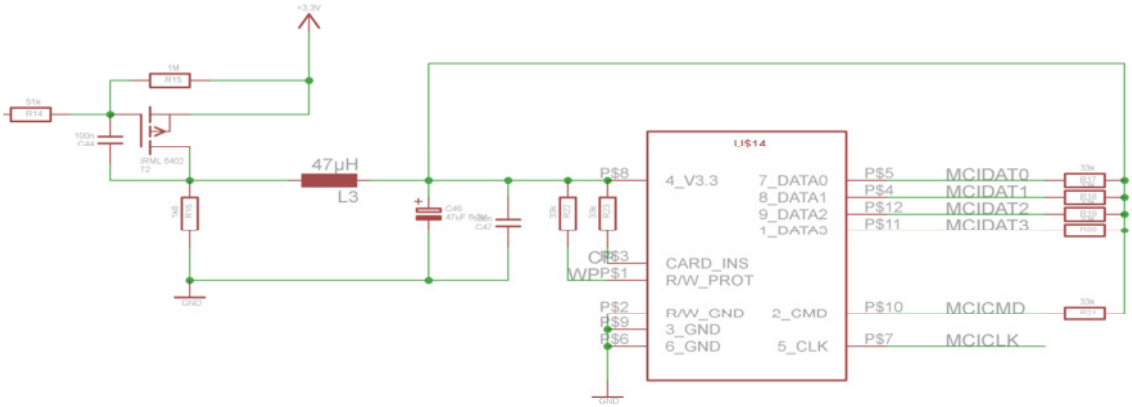

Abbildung 5: SD-Karten Anbindung

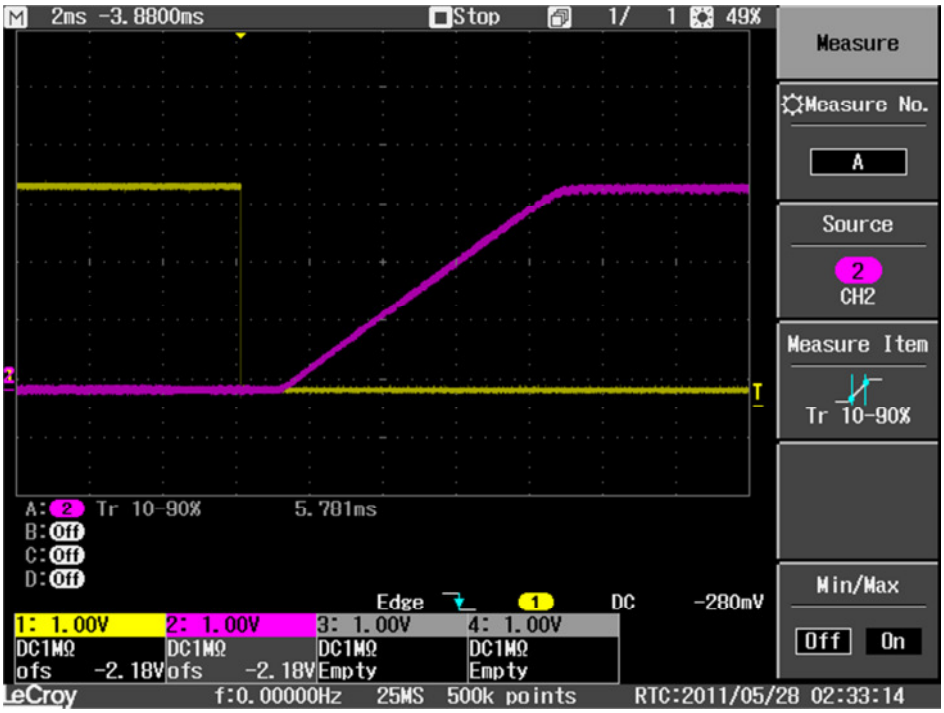

Abbildung 6: Einschalten FET

Hier sieht man die Schaltzeit des FET. Am Kanal 1 (gelb) das Schaltsignal des Prozessors, am Kanal 2 (violett) der Spannungsverlauf auf der SD-Karte.

4.6 *Display Integration*

Das 7“ Display von Admatec wird über ein 40-poliges FFC Kabel verbunden. Im Nachfolgenden sehen Sie eine Tabelle, welche die Verbindungen zwischen Display

und Prozessor wiedergibt. Achtung, die im Folgenden angeführten Leitungen dienen ausschließlich zur Anzeige, die Leitungen für das Touchpanel folgen im nächsten Abschnitt (4.7).
Wie Sie in der Tabelle sehen können, wird das Display mit 16bit Farbinformationen pro Pixel angesteuert. Das entspricht einer Farbauflösung von 65.535 Farben.
Das DE Signal enthält Steuerinformationen für Zeilenanfang und Bildanfang.
Zusätzlich wird noch ein Pixeltakt übertragen, der den Zeitpunkt angibt, ab dem die Farbinformationen gültig sind, übertragen.

Tabelle 2: Display Verbindungen

FFC Pin	Bezeichnung	LPC2478 Pin	Beschreibung
1	Vcc	+3.3V	Versorgungsspannung
2	Vcc	+3.3V	Versorgungsspannung
3	Vcc	+3.3V	Versorgungsspannung
4	Vcc	+3.3V	Versorgungsspannung
5	NC	NC	nicht verbunden
6	DE	P2[4] / LCDENAB	Daten enable (Hier ist HSYNC und VSYNC Codiert)
7	GND	GND	GND
8	NC	NC	nicht verbunden
9	GND	GND	GND
10	NC	NC	nicht verbunden
11	GND	GND	GND
12	B5	P1[29] / LCDVD[23]	Blau 5, MSB
13	B4	P1[28] / LCDVD[22]	Blau 4
14	B3	P1[27] / LCDVD[21]	Blau 3
15	GND	GND	GND
16	B2	P1[26] / LCDVD[20]	Blau 2
17	B1	P2[13] / LCDVD[19]	Blau 1
18	B0	GND	Blau 0, LSB (auf Masse wegen 16 Bit Ansteuerung)
19	GND	GND	GND
20	G5	P1[25] / LCDVD[15]	Grün 5, MSB
21	G4	P1[24] / LCDVD[14]	Grün 4
22	G3	P1[23] / LCDVD[13]	Grün 3

23	GND	GND	GND
24	G2	P1[22] / LCDVD[12]	Grün 2
25	G1	P1[21] / LCDVD[11]	Grün 1
26	G0	P1[20] / LCDVD[10]	Grün 0, LSB
27	GND	GND	GND
28	R5	P2[9] / LCDVD[7]	Rot 5, MSB
29	R4	P2[8] / LCDVD[6]	Rot 4
30	R3	P2[7] / LCDVD[5]	Rot 3
31	GND	GND	GND
32	R2	P2[6] / LCDVD[4]	Rot 2
33	R1	P2[12] / LCDVD[3]	Rot 1
34	R0	GND	Rot 0, LSB
35	NC	NC	NC
36	GND	GND	GND
37	GND	GND	GND
38	DCLK	P2[2] / LCDDCLK	Pixel Clock
39	GND	GND	GND
40	GND	GND	GND

40poliger FFC Stecker Abbildung, veranschaulicht nochmals die Verbindung zwischen Prozessorpins und dem 40poligen Verbindungskabel.

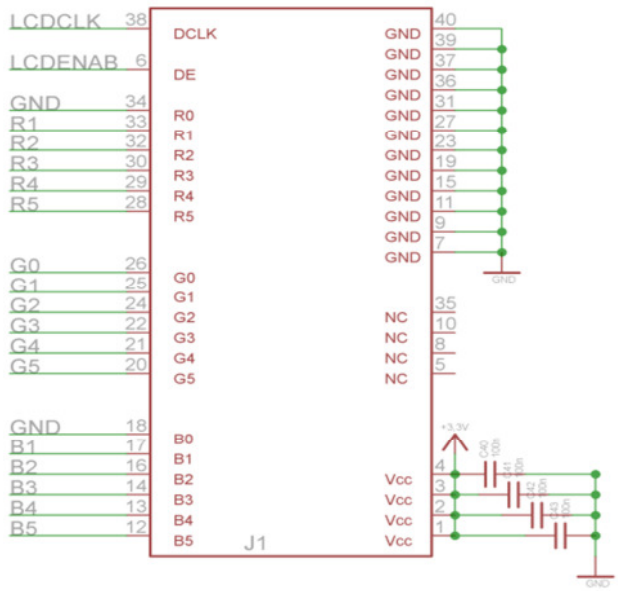

Abbildung 7: LCD-Belegung

4.7 *Auswertung des Touchscreens*

4.7.1 *Grundlagen*

Da es sich bei dem Eingabeinterface um einen 4-wire resistiven Touch handelt, werden 4 Prozessorleitungen zur Auswertung benötigt. Prinzipiell besteht der Touchscreen aus zwei durchsichtigen Widerstandsfolien, die im Ruhezustand voneinander isoliert sind. Wird nun eine Eingabe auf diesem Touchscreen durchgeführt, so entsteht an der jeweiligen Position ein Kontakt zwischen den beiden Folien. Eine dieser Widerstandsfolien besitzt jeweils an den X-Seiten eine Kontaktierung, die andere jeweils an den Y-Seiten. Nun sind diese vier Enden mit dem Prozessor verbunden, wobei jeweils eine X Kontaktierung und eine Y Kontaktierung auf einen ADC Eingang des Prozessors gehen müssen. Die nachfolgende Grafik (Abbildung 8) soll nun das Auswerteverfahren veranschaulichen.

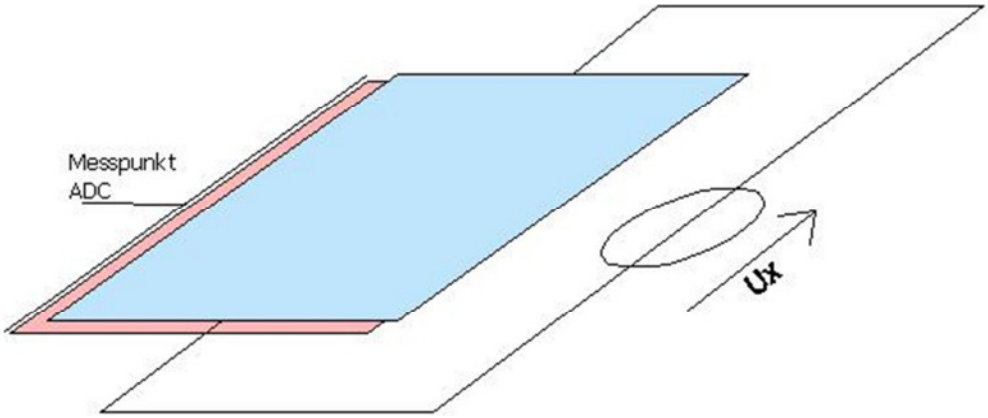

Abbildung 8: Aufbau Touchfolie

Die hier blau eingefärbte Folie soll an den beiden X-Seiten ihre Kontaktierungen besitzen, die rosa eingefärbte Folie an den Y-Seiten. Die Zeichnung und die folgende Erklärung gilt für die Bestimmung der X-Koordinate der Eingabe. Die Bestimmung der Y-Koordinate erfolgt auf die gleiche Weise.

4.7.2 *Positionsbestimmung*

Zur Bestimmung der X-Koordinate werden die Prozessorpins, die mit der blau eingefärbten Folie verbunden sind, als Ausgang geschaltet, die Pins der anderen Fläche auf Tristate, bzw. ein Pin wird als Eingang des ADC definiert. Außerdem wird ein Pin der blauen Fläche auf Vcc, der andere auf GND geschaltet. Das soll in der Grafik durch die Spannungsquelle symbolisiert werden. Danach kann auf der hier rosa eingefärbten Folie mit Hilfe des ADC die X-Entfernung gemessen werden. Da die Widerstandsfolie, an der die Spannung anliegt eine nahezu linearen Spannungsverlauf hat, entspricht die Spannung, die an der Druckposition auf die zweite Folie übertragen wird, der X – Koordinate der Druckposition. Allerdings ist auch die Y – Folie eine Widerstandsfolie und somit gilt diese Rechnung nur, wenn der Eingangsstrom für den ADC mit nahezu Null angenommen werden kann.
Kann dieser Strom nicht mit Null angenommen werden, so kommt es durch den Spannungsabfall an der Y-Folie zu einer Verfälschung. Dieser Messfehler kann aber durch geschicktes Messen ausgeglichen werden.

4.7.3 *Berechnung*

Die nachfolgende Skizze (Abbildung 9) soll zur besseren Erklärung des Messverfahrens dienen. Die blauen Pfeile sollen den Spannungsabfall auf der X-Folie symbolisieren (die eigentliche Messgröße), die rosa Pfeile den Spannungsabfall auf der Y-Folie (Messfehler durch Eingangsstrom des ADC).

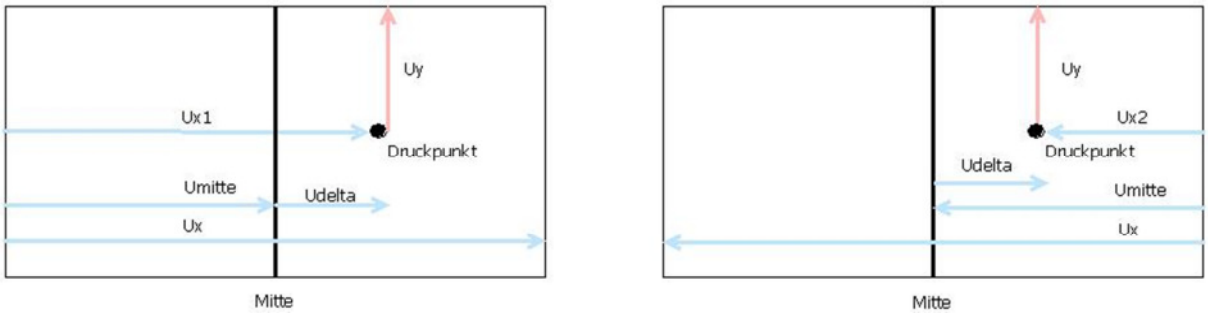

Abbildung 9: Auswertung Touchscreen

Die Messung wird zur Reduktion des Messfehlers auf zwei Messungen erweitert. Die erste Messung (hier links in der Grafik) wird wie zuvor durchgeführt. Dazu wird ein Kontakt der X-Folie auf VCC, der andere auf GND geschaltet und anschließend die Spannung auf der Y-Folie gemessen. Wir nennen den Messwert U1.

Im zweiten Schritt wird die Polarität der X – Folie vertauscht. Das bedeutet, der Pin des Prozessors, der zuvor auf VCC geschaltet war, wird jetzt auf GND geschaltet und umgekehrt. Anders ausgedrückt, der Spannungspfeil in der vorigen Grafik wird umgedreht. Anschließend wird wieder auf der Y-Folie die Spannung gemessen. Diesen Messwert nennen wir U2.

Formel 1: Touchauswertung

$$U_1 = U_{X1} + U_Y \qquad U_2 = U_{X2} + U_Y$$

$$U_{X1} = U_{Mitte} + U_{delta} \qquad U_{X2} = Ux - U_{Mitte} - U_{delta} = U_{Mitte} - U_{delta}$$

$$U_1 = U_{Mitte} + U_{delta} + U_Y \qquad U_2 = U_{Mitte} - U_{delta} + U_Y$$

$$U_1 - U_2 = U_{Mitte} + U_{delta} + U_Y - U_{Mitte} + U_{delta} - U_Y$$

$$U_1 - U_2 = 2 * U_{delta}$$

In Worten ausgedrückt, bedeutet dieses Ergebnis, dass wir durch Subtraktion des Messwertes U2 vom Messwert U1, die doppelte Abweichung von der Mittelposition haben und somit der durch den Eingangsstrom des ADC bedingte Messfehler wegfällt.

4.8 *Externes RAM*

4.8.1 *Anforderungen*

Um das von uns gewählte Display betreiben zu können wird, ein Datenpuffer von 800x480 x 2 Byte benötigt. Das entspricht 750kByte (oder 937,5kByte bei unserem 10.4“ Display). Da der LPC2478 lediglich über maximal 98kByte internes RAM verfügt, mussten wir zusätzlich externes RAM an den Prozessor anbinden.
Zu diesem Zweck entschieden wir uns SDRAM zu verwenden. Diese Art von RAM ist zum einen kostengünstig und zum anderen in größeren Speichergrößen erhältlich. Zusätzlich wollten wir das RAM über die volle Busbreite von 32Bit mit dem Prozessor verbinden, um die Bandbreite zu erhöhen.
Das Problem dabei war allerdings, dass Speicherbausteine mit 16Bit Busbreite wesentlich verbreiteter (einfacher zu beschaffen) und auch preisgünstiger sind. Somit kommt es bei in Summe gleicher Speicherkapazität günstiger, zwei 16Bit Module zu verwenden als ein 32Bit Modul. In diesem Vergleich sind jetzt nicht die zusätzlichen Leiterplatten- und Bestückungskosten berücksichtigt.
Nichts desto trotz entschieden wir uns, in unserem Demonstrationsboard zwei 16Bit breite RAM Module parallel zu schalten und so wieder auf ein 32Bit breites RAM Interface zu kommen, da diese bei einem uns bekannten Distributor sofort für uns verfügbar waren.

4.8.2 *Größe*

Was die Speichergröße betrifft, stellten wir folgende Überlegung an: Wenn wir von dem größeren Display ausgehen, brauchen wir alleine für den Ausgabepuffer ~1MB an RAM. Wenn wir danach auf dem Display zeichnen wollen, noch einmal das selbe für einen Anzeigepuffer, also schon ~2MB. Wollen wir ohne flackern und ohne dass man den Bildaufbau erkennen kann auf dem Display zeichnen, so brauchen wir eine doppelte Pufferung, also noch einmal ein MB dazu, sodass wir schon bei ~3MB sind. Durch diese Überlegungen kamen wir schlussendlich auf eine minimale Speichergröße von 4MB, die wir in unserem Demoboard verbauen mussten.
Bei der Planung und Fertigung unseres Prototypen bestückten wir allerdings Module mit 32 MB, um genug Speicher für diverse Musterapplikationen zu haben.

Später würden sich, wenn die Anforderungen für die Applikation feststehen, diese Module einfach durch kleinere Module ersetzt lassen, da diese pinkompatibel sind. Durch die 32MB Module – und da wir zwei dieser Module parallel geschaltet haben, um die Bandbreite zu erhöhen – ergibt sich eine Gesamtspeicherkapazität unseres externen Rams von 64MB (Achtung, bei RAMs wird häufig die Speicherkapazität in Mega Bit angegeben, das wäre bei uns in Summe 256Mb.).

4.8.3 *Beschaltung*

Im Nachfolgenden (Abbildung 10) ist die Parallelschaltung der SDRAMS und die Anbindung an den Prozessor dargestellt. Wie man in der Abbildung sehen kann, sind die Steuerleitungen (bis auf die Data Mask (DQM) Leitungen) und Adressleitung auf beide SDRAMs parallel geführt. Die Datenleitungen sind natürlich getrennt geführt um die 32Bit Bandbreite zu erzeugen. Natürlich ist jeder Pin, der zur Versorgung dient, mit 100nF gegen Masse abgeblockt.

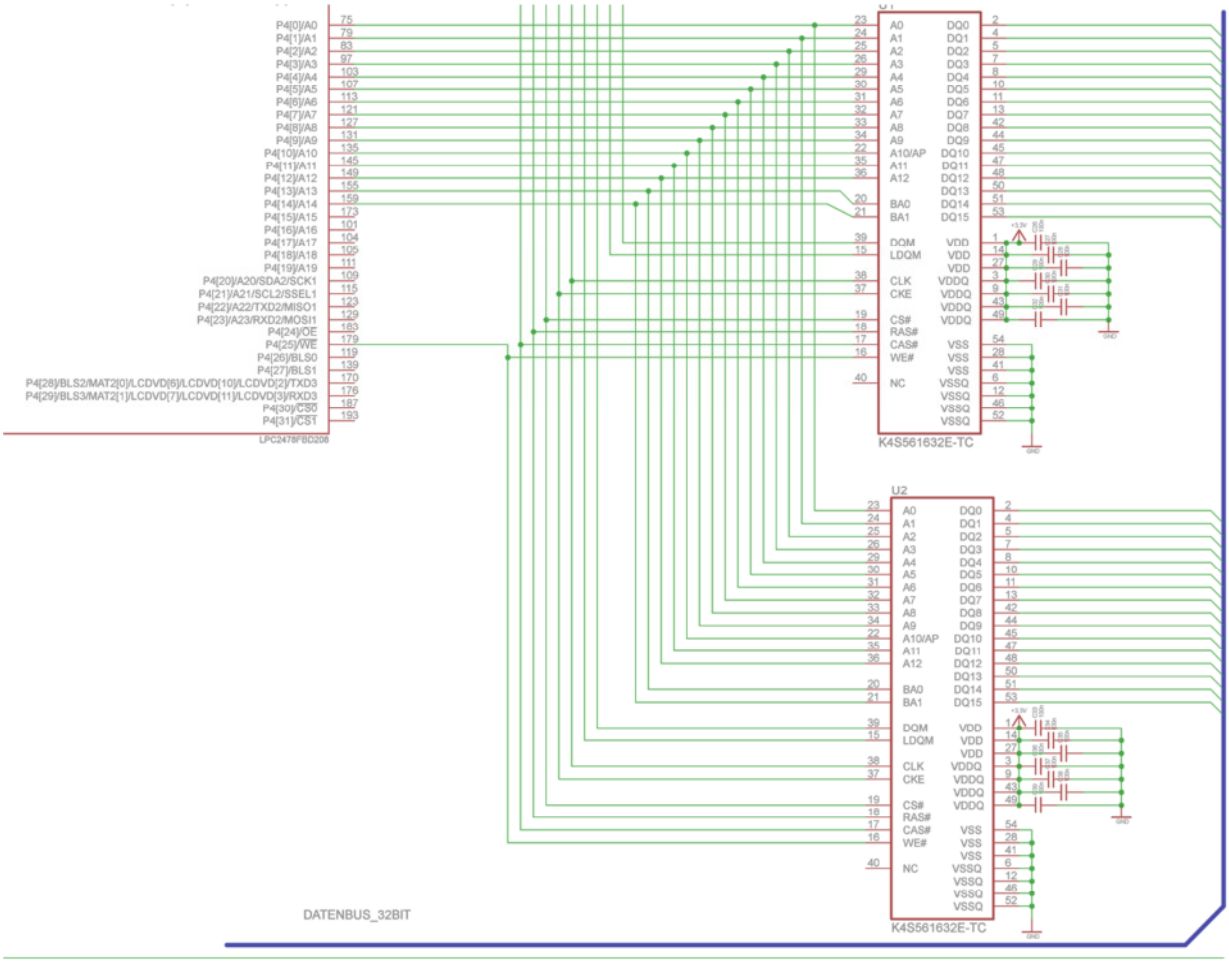

Abbildung 10: Externe RAM Beschaltung

4.9 *CAN – Netzwerk*

4.9.1 *Allgemein*

Wie in der Aufgabenstellung gefordert, soll das Demonstrationsboard über eine CAN-Bus Schnittstelle verfügen. Für diesen Zweck kommt uns der Umstand sehr gelegen, dass der von uns gewählte Prozessor (LPC2478 – NXP) bereits einen CAN-Controller integriert hat, lediglich der Transceiver muss noch extern implementiert werden.

Zu diesem Zweck wurde bei unserem Demoboard nach einem speziellen Transceiver gesucht. Die meisten CAN-Transceiver benötigen eine +5V Versorgungsspannung. Da aber auf unserem Board alle Bauteile mit 3.3V versorgt werden – und wir nicht noch eine zusätzliche Spannungsversorgung aus Kosten- und Platzgründen implementieren wollten – suchten wir nach einem Transceiver, der bei 3.3V betrieben werden kann. Unter diesen Anforderungen wurden wir bei Texas Instruments mit dem SN65HVD231 fündig. Dieser Transceiver benötigt lediglich eine 3.3V Spannungsversorgung, die auf unserem Board ohnedies schon implementiert ist.

4.9.2 *Zusätzlich benötigte Komponenten*

Im nachfolgenden Schaltbild(Abbildung 11) sieht man die Implementierung des CAN-Transceivers. Wie man darin erkennen kann, wurde ein 120 Ohm Widerstand eingeplant, der mit Hilfe eines Lötjumpers zwischen CAN-High und CAN-Low geschaltet werden kann. Dieser Widerstand dient zur Terminierung des Busses (erster und letzter Teilnehmer müssen terminiert werden). Zusätzlich wurde eine Supressordiode eingeplant, um eventuelle Spannungsspitzen am CAN-Bus zu unterdrücken.

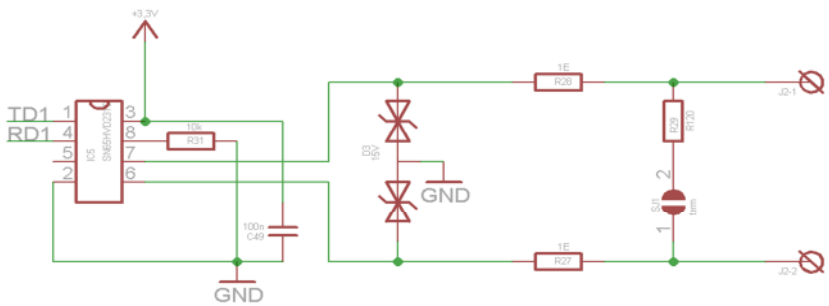

Abbildung 11: CAN Schaltung

5 Entwicklung der Software

5.1 *Einleitung*

5.1.1 *Benötigte Hardwarekomponenten*

Um effektiv an der Software für ein solches Demonstrationsboard arbeiten zu können, war uns klar, dass wir ein Debug Tool benötigen. Prinzipiell lässt sich der Prozessor auch direkt über den integrierten UART programmieren, jedoch ist bei dieser Möglichkeit der Download sehr zeitintensiv und es existieren keine Debugfeatures. Um also komfortabel an der Softwareentwicklung arbeiten zu können, hatten wir bereits beim Schaltungsentwurf darauf geachtet, alle für das Debugen über JTAG notwendigen Steuerleitungen herauszuführen. Zu diesem Zweck planten wir auf unserer Printplatte eine 20 polige 2 reihige Stiftleiste (Rastermaß 2,54mm) ein, welche einen Standard für JTAG bei ARM Prozessoren darstellt.

Um nun mit dem eigentlichen Programmentwurf starten zu können mussten wir uns nur noch für einen Debug Dongle entscheiden. Dieser stellt die Verbindung zwischen der USB Schnittstelle des PC's und dem Jtag Interface des Target Prozessors dar. Solche Debug Dongles sind von diversen Herstellern erhältlich und sind nur bedingt unter verschiedenen Entwicklungsstudios kombinierbar.

Weiters geht auch die Preisspanne deutlich auseinander, so kostet beispielsweise der Debug Dongle einer sehr bekannten und renommierten Firma ~1.000€ (Stand April 2011), ein Noname Produkt weniger als 50€. Wir möchte an dieser Stelle ausdrücklich darauf hinweisen, dass natürlich auch der Funktionsumfang (Debugmöglichkeiten, Geschwindigkeit, Kompatibilität…) durchaus variiert.

Um unser Budget in Grenzen zu halten, versuchten wir im unteren Preissegment fündig zu werden. Ein weiteres Kriterium für die Wahl unseres Debug Tools war, dass wir versuchten, einen Debug Dongle zu finden, der mit OpenOCD funktioniert. OpenOCD ist eine Serveranwendung zum Debuggen, auf die wir später noch zurückkommen werden. Unter diesen genannten Vorraussetzungen konnten wir einen Debugger finden, der unsere Wünsche sehr gut erfüllte. Dabei handelt es sich

um den ARM-USB-OCD von Olimex für 54,95€[1]. Dieser Debug Dongle besitzt einen USB Anschluss für die Verbindung mit dem PC, einen 20 poligen JTAG – Verbinder, wie er auch auf unserem Demonstrationsboard vorgesehen ist, und zusätzlich noch eine serielle Schnittstelle, die als USB-RS232 Konverter genutzt werden kann.
Weiters wird laut Produktbeschreibung zusätzlich noch eine CD mitgeliefert, welche Open Source Entwicklungstools als Alternative zu kommerziellen Entwicklungsstudios beinhaltet.
Nachdem unsere Entscheidung feststand, bestellten wir den Debug Dongle, um mit der eigentlichen Softwareentwicklung endlich starten zu können.

5.1.2 *Benötigte Softwarekomponenten*

5.1.2.1 *Open Source Tools*

Nachdem wir unseren JTAG – Dongle geliefert bekamen, versuchten wir eine Open Source Debug Plattform auf einem Microsoft Windows XP Rechner zu installieren.
Die freie Plattform setzt sich grundsätzlich aus mehreren Komponenten zusammen: Zum Ersten der OCD (On Chip Debug) Server. Diese Serveranwendung verbindet sich mit dem JTAG Dongle und stellt das Interface zwischen PC und JTAG-Interface dar.
Zum Zweiten die GNU ARM Toolchain. Diese Sammlung enthält C/C++ Kompiler, Linker und Debugger und zum Dritten Eclipse als IDE (Integrated Developement Enviroment). Die nachfolgende Grafik (Abbildung 12) soll den Zusammenbau etwas veranschaulichen.

[1] Stand 15.04.2011, Olimex Webshop

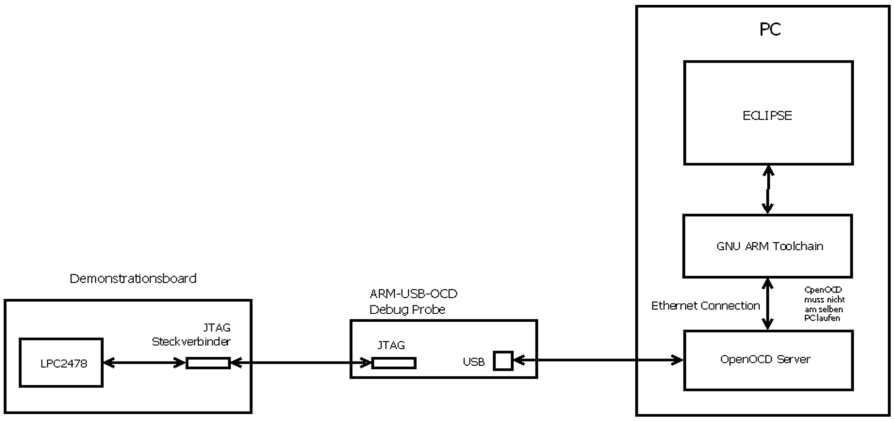

Abbildung 12: OpenOCD Komponenten

Der OpenOCD Server muss nicht am selben PC laufen, auf dem die Software geschrieben wird, da sich die GNU ARM Toolchain über eine Ethernet Verbindung mit dem OCD Server verbindet. Weiters wird der OCD Server über Textdateien konfiguriert, um eine Verbindung mit dem Target herzustellen (JTAG Geschwindigkeit, Target Prozessor Typ,…). Wir hatten über mehrere Wochen mit diesem Versuchsaufbau getestet und konnten auch Programme in den Flash-Speicher des Prozessors unseres Demonstrationsboardes programmieren. Das Problem dabei war jedoch, dass dieser Versuchsaufbau nie stabil lief. Einmal gab der OCD Server die Fehlermeldung aus, dass dieser nicht mit dem Dongle kommunizieren kann, ein anderes Mal, dass kein Target gefunden werden kann und das nächste Mal wieder konnte die Toolchain nicht auf den Server zugreifen. Die prozentuelle Entwicklungszeit, die wir für den Betrieb des „Debugger“ benötigten war weit über 50% im Vergleich zum Codeentwurf.

5.1.2.2 *Kommerzielle Tools*

Aus diesen oben genannten Gründen entschlossen wir uns nach kommerziellen und hoffentlich stabileren Lösungen zu suchen. Olimex wies in der Produktbeschreibung darauf hin, dass ihr Debugger mit dem Entwicklungsstudio von IAR embedded workbench kompatibel sein soll. Nach kurzer Recherche stellten wir jedoch fest, dass auch dieses Entwicklungsstudio lediglich über den OpenOCD Server mit dem Dongle kommunizieren kann.
Also starteten wir eine Internetrecherche nach Entwicklungsstudios, die unsere Debugger in einer nativeren und besseren Weise unterstützen. Wir wollten es nicht

glauben, dass unser Debug Tool nicht von einem Entwicklungsstudio brauchbar unterstützt wird.
Nach einiger Recherche stießen wir auf das Entwicklungsstudio von Rowley, welches mit einer nativen Unterstützung des ARM-USB-OCD wirbt. Rowley bietet weiters den Gratisdownload einer Testversion seines Entwicklungsstudios CrossWorks an. Nach kurzem Studium der Lizenzbediengungen stellten wir zudem fest, dass Rowley eine Lizenz für unsere Zwecke für ~150$ anbietet. Also luden wir uns erstmal die Testversion aus dem Internet und installierten diese. Zu unserem Erstaunen wurde der Debuger sofort gefunden. Zusätzlich wurden zu dem Entwicklungsstudio Muster Konfigurationsfiles für unseren Zielprozessor mitgeliefert. Nachdem wir einige Einstellungen für die Debug Probe machten (Taktrate für JTAG Verbindung, Prozessorclock, …) konnten wir uns tatsächlich sofort mit unserem Demonstrationsboard verbinden. Nun konnten wir uns also an die Arbeit machen, den Zielprozessor selbst zu konfigurieren und die Firmware für die Targetplattform zu programmieren.

5.2 *Prozessorinterner Aufbau*

5.2.1 *Allgemein*

Nachdem wir uns nun für ein Entwicklungsstudio (Rowley Cross Studio) entschieden hatten und eine lauffähige Instanz davon installiert hatten, konnten wir uns der Softwareentwicklung widmen. Weiters hatten wir zu diesem Zeitpunkt auch schon unseren Prototypen fertigen lassen und konnten so, mit Hilfe unseres Debugers (ARM-USB-OCD) direkt auf unserer Zielplattform entwickeln.
Mit unserem Entwicklungsstudio wurden einige Code-Segemente mitgeliefert, um den Prozessor zu konfigurieren und mit dem Debugger zu kommunizieren. Nun galt es, dieses als erstes zu konfigurieren. Bevor wir nun auf dessen Konfiguration eingehen, möchten wir noch kurz etwas zur Organisation des Flash erwähnen.

5.2.2 *Organisation des Flashspeichers*

Der Flash Speicher unseres Prozessors ist in so genannte Sektoren unterteilt. Die einzelnen Sektoren haben unterschiedliche Größen. In Summe gibt es 28 Sektoren. Die Sektoren 0 bis 7 haben 4kB, die Sektoren 8 bis 21 haben 32 kB, und die

Sektoren 22 bis 27 wieder 4kB. Weiters besitzt der Prozessor einen so genannten Primary Bootloader.

5.2.3 *Primary Bootloader*

Mit diesem ist es möglich, beim Power up über den UART0 direkt in den Flash Speicher zu schreiben. Dazu wird beim Einschalten der Versorgungsspannung des Prozessors (oder bei einem Reset) von diesem selbst ein Port gesampelt (P2.10). Ist dieses beim Start „low“ startet der Prozessor eine Auto Baud Rate Detection und wartet auf Daten über den UART. Weiters existiert durch diesen Bootloader aber auch die Möglichkeit, aus der Firmware heraus ebenfalls Sektoren zu löschen und zu beschreiben.
Der Grund warum wir hier auf die interne Speicherstruktur eingehen, wird im Kapitel Softwareupdate ersichtlich.
Prinzipiell startet der Prozessor nach einem Power up oder einem Reset – sofern er nicht durch das externe Port in den Programmiermodus gezwungen wird – an der Adresse 0x0000 0000 (dem Reset Handler mit der Ausführung des Programmcodes). Das ausführliche Schreiben der Adresse soll veranschaulichen, dass die Adressierung – ebenso wie alle Register des Prozessors – 32Bit breit sind. Um nun die Konfiguration und den Startup erklären zu können, müssen wir zuerst noch einige Hardwaredetails zum Prozessor erörtern.

5.2.4 *Pipeline*

Der CPU Kern des LPC2478 verfügt über eine dreistufige Pipeline. In der ersten Stufe wird der Befehl geladen, in der zweiten decodiert und der dritten ausgeführt. Durch diese Pipeline kann die Abarbeitungsgeschwindigkeit des Prozessors deutlich gesteigert werden. So können damit die meisten ARM Befehle in nur einem Taktzyklus abgearbeitet werden. Ein Problem entsteht hier natürlich bei Sprüngen, da in diesem Fall die Pipeline geleert wird, und mit neuen Befehlen gefüllt werden muss. Um diesem Effekt entgegenzuwirken sind im ARM Befehlssatz Befehle mit Bedingungen implementiert. So wird zum Beispiel beim Befehl EQADD R1, R2, #0x0000 0005 die Summe vom Register R2 +5 nur dann berechnet und auf das Register R1 geschrieben, wenn die vorige Berechnung das „equal“ Flag gesetzt hat. Ist dies nicht der Fall, so wird die Berechnung durch ein NOP (No Operation) ersetzt.

Diese bedingten Befehle haben den Vorteil, dass im Fehlerfall nicht die gesamte Pipeline geleert wird, sondern lediglich ein Taktzyklus „verloren“ geht.

5.2.5 *Befehlssatz*

Im vorigen Absatz haben wir vom ARM – Befehlssatz gesprochen. Der Grund dafür ist, dass der Prozessor über zwei Befehlssätze verfügt. Den ARM-Befehlssatz und den Thumb-Befehlssatz. Der gravierendste Unterschied ist, dass der ARM-Befehlssatz 32-, der Thumb dagegen nur 16 Bit breit ist. Etwas präziser gesprochen, benötigt der Thumb-Befehlssatz weniger Programmspeicher für den selben Algorithmus, der ARM-Befehlssatz ist dafür schneller. Um das Ganze noch mit Zahlen zu unterstreichen ist der ARM-Befehlssatz um etwa 40% schneller und der Thumb-Befehlssatz produziert einen um etwa 30% kleineren Code.
Der Thumb – Befehlssatz unterstützt weiters keine bedingten Befehle (außer natürlich bedingte Sprünge) und auch manche „Spezialbefehle“ werden nicht unterstützt. Auf diese werden wir gleich noch zurückkommen. Da es gewisse Befehle gibt, die nicht von diesem Befehlssatz unterstützt werden, gibt es also Funktionen, die dezidiert mit dem ARM – Befehlssatz kompiliert werden müssen.

5.2.6 *Prozessorinterne Register*

Um den Startvorgang erläutern zu können, ist noch ein weiteres Detail nötig. Prinzipiell hat der Prozessor 15 Register, den Program Counter (PC – gibt die aktuelle Position im Programmspeicher wieder) und ein Current Program Status Register (CPSR). Von diesen 15 Registern sind die Register R0 bis R12 generelle Register, das Register R13 der Stack Pointer und R14 das Link Register (hier wird bei Sprüngen die Rücksprungadresse gespeichert). Doch das ist noch nicht alles. Bei außergewöhnlichen Ereignissen (welche das sein können werden wir im Folgenden erörtern) kann der Prozessor seinen so genannten Modus ändern. So gibt es bei dem Prozessor insgesamt 6 Modes.

5.2.7 *Modes*

- User Mode: Hier wird der „normale“ Code ausgeführt.
- FIQ: Fast Interrupt Request
- Supervisor: Zugriff auf spezielle Register, beim Programmstart

- Abort: Ungültiger Zugriff auf Daten oder Code
- IRQ: Interrupt Request
- Undefined: Bei ungültigem Befehl

Jeder dieser Modes, mit Ausnahme des User Modes, hat ein zusätzliches Register – das SPSR Register. Hier wird bei einem Mode Wechsel der Inhalt des CPSR gesichert. Dies ermöglicht das Debuggen und Analysieren bei einem solchen Wechsel. Weiters haben diese Modes alle ein eigenes R13 und R14 (Stack Pointer und Link Register). Alle anderen Register müssen – wenn diese im jeweiligen Mode verwendet werden – zuerst auf den Stack gesichert und anschließend wiederhergestellt werden.
Eine weitere – und nun auch letzte Ausnahme – stellt der FIQ Mode dar. Dieser hat zusätzlich noch eigene R7 bis R12 Register. Dies ermöglicht eine sehr schnelle Reaktion auf Interrupts, ohne zuerst Register sichern zu müssen.

5.2.8 *Current Programm Status Register*

Nun haben wir schon des Öfteren das Current Programm Status Register (kurz CPSR) erwähnt. Daher noch eine kurze Erklärung, wozu dieses benötigt wird und wie man darauf zugreifen kann.

Tabelle 3: Current Programm Status Register

CPSR - Register	
Bit	Bezeichnung
0	Mode
1	Mode
2	Mode
3	Mode
4	Mode
5	Thumb Befehlssatz
6	Fast Interrupt enable
7	Interrupt enable
28	Overflow flag
29	Carry flag
30	Zero flag
31	Negative flag

Die Bits 0-4 des CPSR Registers geben den aktuellen Mode des Prozessors wieder (User, FIQ, Supervisor,..), das Bit 5 kennzeichnet in welchem Befehlssatz (ARM oder

THUMB) sich der Prozessor gerade befindet. Dieses Bit darf aber nicht direkt gesetzt werden. Um den Befehlssatz des Prozessors zu ändern, gibt es einen eigenen Sprungbefehl, der zugleich den Befehlssatz von ARM auf THUMB oder vice versa ändert (BX). Weiters wird der Befehlssatz auch automatisch geändert, wenn eine Ausnahme auftritt.
Die Bits 5 und 6 dienen dazu, die Interrupts am Prozessor zu aktivieren oder zu deaktivieren.
Die Bits 28 bis 31 werden auch Condition flags genannt. Tritt beispielsweise bei einer Addition zweier Register ein Überlauf auf, so wird das Bit 28 – das Overflow Bit – gesetzt. Der nächste Befehl könnte nun zum Beispiel ein bedingter Befehl sein, der nur ausgeführt wird, wenn das Overflow Bit nicht gesetzt ist (z.B. NVMOV R1, #0x0000 0005 – würde jetzt nur dann den Wert 5 auf das Register R1 schreiben, wenn die vorherige Operation nicht das Overflow Bit gesetzt hätte).
Eine weitere wichtige Besonderheit bei diesem Register gibt es noch. Auf dieses Register lässt sich nur mit zwei speziellen Befehlen aus dem ARM Befehlssatz zugreifen. Mit MSR kann der Inhalt des CSPR auf ein generelles Register kopiert werden und mit MRS der Wert von dem generellen Register zurück auf das CSPR Register gespeichert werden.
Soll nun z.B. der Interrupt des Prozessors aktiviert werden, so muss das CSPR Register auf beispielsweise R0 kopiert werden, auf R0 kann das Bit 7 (IRQ enable Bit) gelöscht werden und anschließend der neue Inhalt zurück auf das CSPR Register geladen werden.

5.2.9 *Ausnahmen*

Kommen wir nun auf die Modes zurück. Der Grund dafür, dass es überhaupt diese unterschiedlichen Modes gibt, ist, dass es Ausnahmen gibt, die den Prozessor aus dem normalen Betrieb heraus unterbrechen.
So sind sieben Ausnahmen definiert, bei denen der Prozessor die „normale“ Code Ausführung unterbricht, an einen definierten Punkt im Programmcode springt und den Mode ändert.
Diese sieben Ausnahmen sind: Reset, Undefined Instruction, Software Interrupt, Prefetch Abort, Data Abort, IRQ und FIQ.

Um diese Ausnahmen im Programm bearbeiten zu können sind Einsprungadressen definiert. Das ist auch der Grund, warum diese hier so ausführlich erläutert werden . Bei der Systeminitialisierung müssen diese nämlich definiert werden.
Da wir den Prozessor in der Programmiersprache C mit entsprechendem Compiler programmieren, müssen wir uns während des Betriebs keine Gedanken darüber machen, Register auf den Stack zu sichern und wiederherzustellen. Trotzdem ist es wichtig zu wissen, was im Hintergrund passiert, da wir beim Systemstart z.B. die Einsprungadressen für die Ausnahmen definieren müssen.

5.2.10 *Einsprungadressen*

Wie bereits erwähnt, startet der Prozessor LPC2478 von NXP nach einem Reset an der Einsprungadresse 0x0000 0000. An dieser Stelle steht dann für gewöhnlich ein Sprung an die Adresse, wo sich der eigentliche Programmcode befindet. Zu Beginn des Flash Sektors 0 befindet sich aber nicht nur die für den Reset zuständige Einsprungstelle – hier sind auch die Einsprungstellen für alle weiteren Ausnahmen definiert.

Tabelle 4: Einsprungvektoren bei Ausnahmen

Adresse	Ausnahme
0x0000 0000	Reset
0x0000 0004	Undefined Instruction
0x0000 0008	Software Interrupt
0x0000 000C	Prefetch Abort
0x0000 0010	Data Abort
0x0000 0014	Spezielle Bedeutung
0x0000 0018	IRQ
0x0000 001C	FIQ

Wie aus der Tabelle ersichtlich ist, befindet sich an der Adresse 0x0000 0014 keine Einsprungadresse. Dafür hat diese Adresse ein spezielle Bedeutung. Wie wir zu Beginn des Hardwareabschnittes bereits erklärt haben, beginnt der Prozessor nach einem Reset als Erstes einen Portpin auszuwerten, um zu entscheiden, ob an der Adresse 0x0000 0000 gestartet werden soll, oder ob er in den Programmiermodus gehen soll, um auf dem UART auf Daten zu warten. Ist die Bedingung auf dem Portpin nicht erfüllt, so gibt es noch ein weiteres Kriterium, welches erfüllt sein muss,

bevor der Prozessor mit der Codeausführung beginnt. Für dieses Kriterium wird getestet, ob sich ein gültiger Programmcode im Flashspeicher befindet. Dafür kommt nun die Flashadresse 0x0000 0014 ins Spiel. Diese stellt nämlich eine Prüfsumme über alle Einsprungadressen dar. Zur Berechnung dieser Prüfsumme werden alle Einsprungadressen zusammengezählt und von den unteren vier Byte das 2er Kompliment gebildet (in der Binärdarstellung der Zahl alle Bits invertieren und eins dazu addieren). Dieses Ergebnis muss an der Adresse 0x0000 0014 gespeichert werden. Beim Starten des Programms addiert der Prozessor ebenfalls alle Einsprungadressen (inklusive der Prüfsumme) und das Ergebnis muss 0x0000 0000 ergeben. Nur dann ist für den Prozessor der Inhalt des Programmspeichers gültig und er beginnt mit der Programmausführung. Ist das Ergebnis ungültig geht er ebenfalls in den Programmiermodus und wartet auf Daten auf der seriellen Schnittstelle.
Daher ist beim Erstellen oder Ändern der Einsprungadressen immer darauf zu achten, auch die Prüfsumme zu aktualisieren, da sonst das neue Programm nicht ausgeführt werden kann.

Um nun die grundlegenden Einstellungen des Prozessors richtig vornehmen zu können und die Einsprungadressen an die richtige Position im Flash zu schreiben, stellen sowohl NXP selbst, als auch die Hersteller der Entwicklungsstudios einen Startupcode zur Verfügung. Dies ermöglicht einen einfachen Einstieg in die doch etwas komplexere Grundkonfiguration des Prozessorkerns.
In unserem Projekt verwendeten wir daher als Grundlage den Startupcode unseres Entwicklungsstudios, den Rowley zur Verfügung stellt, und passten diesen, soweit wie notwendig, an unser Demonstrationsboard an.
Der nachfolgende Codeausschnitt aus dem Startupfile zeigt die Konfiguration der Einsprungadressen.
Der Grund, warum in diesem Codeausschnitt der Load Register Befehl verwendet wird, ist, dass damit der gesamte Adressbereich des Prozessors abgedeckt werden kann. Die „-8“ entstehen durch die Pipeline, da immer schon ein weiterer Befehl decodiert und noch ein weiterer Befehl geladen wird. Somit braucht durch diese Art der Adressierung auch die Prüfsumme für ein gültiges Programm im Flashspeicher nicht geändert werden, wenn sich zum Beispiel die Adresse für den reset_handler ändert.

```
_vectors:
  ldr pc, [pc, #reset_handler_address - . - 8]  // Einsprungadresse Reset
  ldr pc, [pc, #undef_handler_address - . - 8]  // undefined instruction
  ldr pc, [pc, #swi_handler_address - . - 8]    // swi interrupt
  ldr pc, [pc, #pabort_handler_address - . - 8] // abort prefetch
  ldr pc, [pc, #dabort_handler_address - . - 8] // abort data
  .word 0xB9206E58                              // checksumme, gültige Vektoren
  ldr pc, [pc, #-0x120]                        /* Einsprung IRQ
  ldr pc, [pc, #fiq_handler_address - . - 8]    //einsprung FIQ


reset_handler_address:
  .word reset_handler
undef_handler_address:
  .word undef_handler
swi_handler_address:
  .word swi_handler
pabort_handler_address:
  .word pabort_handler
dabort_handler_address:
  .word dabort_handler
fiq_handler_address:
  .word fiq_handler
```

5.2.11 *Interrupts*

Eine Ausnahme stellt die Einsprungadresse für den Interrupt dar. Auf diese werden wir nun eingehen.

Prinzipiell verfügt der Prozessor über zwei Interruptausnahmen. Die IRQ und die FIQ Ausnahme. Weiters sind alle Interruptquellen (insgesamt 32) mit dem Vectored Interrupt Controller (VIC) verbunden. Durch die Konfiguration des Vectored Interrupt Controllers kann jede Interruptquelle aktiviert/deaktiviert und als IRQ oder FIQ definiert werden.

Um einen Interrupt zu aktivieren muss zum einen die Peripherie so konfiguriert werden, dass diese einen Interrupt erzeugt, und zum anderen müssen am VIC folgende Einstellungen gesetzt werden:

VICIntEnable: Mit diesem Register muss der jeweilige Interrupt (0-31) freigegeben werden.

VICIntSelect: Mit diesem Register kann eingestellt werden, ob es sich bei der jeweiligen Quelle um einen IRQ oder einen FIQ handeln soll.
Vector Address Registers 0-31: In diese Register wird – sofern eine Interruptquelle als IRQ definiert ist – die Adresse der Funktion eingetragen, die beim Auftreten eines Interrupts ausgeführt werden soll.
Vector Priority Registers 0-31: Hier kann – sofern eine Interruptquelle als IRQ definiert ist – die Priorität (0 – 15, wobei 15 die niedrigste Priorität darstellt) eingestellt werden. Haben zwei Interruptquellen dieselbe Priorität, so wird der Interrupt mit der niedrigeren Addressnummer bevorzugt behandelt.
Vector Address Register: Dieses Register muss nicht konfiguriert werden, sondern hier steht im Falle eines IRQ die Adresse der Funktion (aus Vector Address Register 0-31) die aufgerufen werden soll.

5.2.11.1 *FIQ*

FIQ ist die schnellstmögliche Interruptbehandlung. Den Grund dafür haben wir schon bei den Modes erörtert. Im FIQ Mode existieren nämlich eigene Register R7-R12. Damit ist es möglich auf Ereignisse sehr schnell zu reagieren, da die Registerinhalte nicht erst auf den Stack gespeichert werden müssen. In einem System sollte, wenn irgendwie möglich, nur ein FIQ definiert werden – auch wenn sich mehrere Interruptquellen gleichzeitig als FIQ definieren lassen. Nur so kann die Antwortzeit des Systems auf ein Ereignis deterministisch bestimmt werden. Außerdem würde die Definition von mehreren Interruptquellen als FIQ weiters dazu führen, dass in der Interrupt Service Routine noch zusätzlich abgefragt werden müsste, um welchen Interrupt es sich handelt, damit das Programm richtig darauf reagieren könnte. Damit würde wieder die Antwortzeit verlängert werden.
Tritt also ein FIQ auf, unterbricht der VIC sofort die Ausführung des aktuellen Programms, wechselt in den FIQ Mode und springt an die Einsprungadresse 0x0000 001C für den FIQ.
Um dem Compiler mitzuteilen, dass es sich bei einer Funktion um einen Fast Interrupt handelt, muss die Funktion mit __fiq gekennzeichnet werden (die Kennzeichnung kann compilerabhängig sein). Damit weiß der Compiler außerdem, wie die Rücksprungadresse berechnet werden muss.

Zur Erklärung: Je nachdem welche Ausnahme aufgetreten ist, muss die Rücksprungadresse unterschiedlich berechnet werden. So muss bei einem IRQ oder FIQ als Rücksprungadresse der Inhalt des Link Register R14 – 4 geladen werden, da bei einem Interrupt der aktuell ausgeführte Befehl unterbrochen wird. Hier ein möglicher Prototyp für eine Fast Interruptfunktion.

```
void Fast_Interrupt (void) __fiq;
```

5.2.11.2 *IRQ*

Tritt ein IRQ auf, so sieht der Prozessablauf des VIC ähnlich wie bei einem FIQ aus. Zuerst wird – falls mehrere Interrupts gleichzeitig auftreten – an Hand der Priorität festgelegt, welcher Interrupt zuerst abgearbeitet werden soll. Die Adresse der Funktion, dessen Interrupt zur Abarbeitung an die Reihe kommt, wird auf Vector Address Register (0xffff ff00) gespeichert. Weiters wird der Prozessor unterbrochen, in den IRQ Mode gebracht und die Codeausführung an der Adresse 0x0000 0018 (Einsprungadresse für IRQ) fortgeführt.
An der Adresse 0x0000 0018 haben wir folgende Codezeile stehen:

```
ldr pc, [pc, #-0x120]
```

Das bedeutet, dass auf den PC der Wert geschrieben wird, der auf PC - 0x120 steht. Da dieser Befehl an der Adresse 0x0000 0018 steht – und der PC ja durch die Pipeline schon um 8 weiter ist – bedeutet das, dass der Wert von der Adresse (0x0000 0018 + 0x08) - 0x120 – also 0xffff ff00 – auf den Programmcounter geschrieben wird. Das genau ist die Adresse des Vector Address Registers, wo sich zu diesem Zeitpunkt die Adresse der Funktion befindet, dessen Interrupt abgearbeitet werden soll.
Das bedeutet also, dass die Ausführung direkt in die Interruptfunktion springt dessen Adresse im VIC eingetragen wurde. Die C-Funktion muss ähnlich wie beim FIQ wieder gekennzeichnet werden (diesmal mit __irq) um dem Compiler wieder mitzuteilen, dass es sich dabei um eine Interruptfunktion handelt.
Am Ende der Funktion muss noch schreibend auf das Vector Address Register zugegriffen werden, um dem VIC zu signalisieren, dass der Interrupt fertig abgearbeitet ist. Das bedeutet also, dass irgendein Dummy-Wert (z.B. 0) auf das Register geschrieben werden muss.

Natürlich muss zuerst auch immer der jeweilige Interrupt in der Peripherie zurückgesetzt werden, da dieser sonst sofort wieder einen Interrupt auslösen würde. Dies gilt sowohl für IRQ als auch für FIQ.

5.3 *Memory Accelerator Module*

5.3.1 *Hintergrund*

Da wir unseren Prozessor – für ein Embedded System – mit einer relativ hohen Taktrate takten (72 Mhz) ist die Zugriffzeit des internen Falshspeichers nicht unwesentlich. Das interne Flash des LPC2478 besitzt eine Zugriffszeit von 50ns. Das würde mit f = 1 / T eine maximale Taktfrequenz von 20MHz ergeben. Eine Alternative, die ebenfalls von unserem Prozessor unterstützt werden würde, ist, den auszuführenden Code zuerst ins interne RAM zu kopieren und von dort auszuführen. Das interne RAM hat eine wesentlich geringere Zugriffszeit und würde die Codeausführung somit natürlich beschleunigen. Das Problem dabei ist jedoch, dass das interne RAM in seiner Größe doch sehr beschränkt (max. 98kB) ist und diese Lösung so für größere Programme nicht in Frage kommt.
Eine andere Möglichkeit, die uns der Prozessor bietet, dieses Problem zu lösen, ist das Memory Accelerator Module (MAM). Dieses versucht ähnlich wie ein Cache die benötigten Daten für den Prozessor (sowohl Code als auch Daten) schon im Vorfeld aus dem Flash zu laden und dem Prozessor bereitzustellen.

5.3.2 *Einstellungen*

Grundsätzlich benötigt dieses Modul lediglich zwei Einstellungen. Zum einen ein Timing Register, welches das Verhältnis von Taktrate des Prozessors und Zugriffszeit des Flashes darstellt, und zum anderen ein Moderegister, welches das MAM in seinem Funktionsumfang konfiguriert.
Prinzipiell kann das MAM mit dem Moderegister ein- oder ausgeschaltet werden. Der Grund, warum das MAM überhaupt ausgeschaltet werden kann ist, dass nur so eine genaue Zeitberechnung bei Sprüngen im Programmcode möglich ist. Muss also ein Programm absolut echtzeitfähig sein, so kann das MAM abgeschaltet werden.
Weiters besteht aus genau diesem Grund auch die Möglichkeit, über das Moderegister des MAM nur teilweise zu verwenden. In diesem Fall werden sequentielle Code Segmente mit Hilfe des Moduls beschleunigt, Sprünge und konstante Daten bleiben aber im Falsh und müssen direkt geladen werden.
Da wir in unserer Applikation keine solchen zeitkritischen Abläufe haben, bei denen wir absolute Echtzeitfähigkeit voraussetzen müssen, können wir das MAM in seinem

vollen Funktionsumfang aktivieren. Dazu schreiben wir auf das MAM_mode_control Register den Wert 0x02, um die volle Funktionalität zu nutzen und auf das MAM_fetch_cycle_timing Register den Wert 0x04. Dieser ergibt sich durch die Zykluszeit unseres Prozessors von 13.8ns und, dass das Flash eine Zykluszeit von 50ns besitzt. Daher sind für einen Zugriff vier Taktzyklen notwendig.

Diese beiden Reigster werden ebenfalls im Startupcode gesetzt.

```
mov r1, #0
str r1, [r0, #MAMCR_OFFSET]
mov r1, #0x04
str r1, [r0, #MAMTIM_OFFSET]
mov r1, #0x02
str r1, [r0, #MAMCR_OFFSET]
```

Die Konstante MAMTIM_VAL wurde von uns mit 0x04 initialisiert, die Konstante MAMCR_VAL mit 0x02.

5.4 *Oszillator*

5.4.1 *Allgemein*

Prinzipiell existieren beim LPC2478 von NXP drei unabhängige Oszillatoren:
Der interne Oszillator, der Main-Oszillator sowie der RTC-Oszillator.

5.4.2 *Interner Oszillator*

Beim internen Oszillator handelt es sich um einen RC Oszillator mit einer nominalen Frequenz von 4MHz. Nach einem Systemstart oder Reset läuft der Prozessor immer von dieser Clock weg. Dies ermöglicht es auch dem internen Bootloader auf einer definierten Frequenz auf der seriellen Schnittstelle zu operieren. (Achtung, der Bootloader konfiguriert, falls er aktiv wird, selbst die PLL, daher muss diese, bevor sie konfiguriert wird, sicherheitshalber deaktiviert werden, da es sonst zu Problemen beim debuggen kommen kann).
Weiters könnte dieser Oszillator auch für den normalen Betrieb des Prozessors verwendet werden. Das Problem jedoch ist, dass dieser nicht sehr genau ist, und NXP im Datenblatt ausdrücklich darauf hinweist, dass dieser Oszillator ungeeignet ist, falls der CAN Controller des Prozessors verwendet werden soll (zumindest bei Taktraten von >100kHz am CAN Bus). Aus diesem Grund entschieden wir uns beim Hardwareentwurf, einen eigenen Oszillator auf unseren Print zu setzen.

5.4.3 *Main Oszillator*

Aus den bereits erwähnten Gründen entschieden wir uns, einen Main Oszillator auf unserem Print zu installieren. Prinzipiell kann dafür ein beliebiger Oszillator oder Schwingquarz verwendet werden, der in einem Frequenzbereich von 1 bis 24 MHz schwingt. Diese Frequenz kann entweder direkt verwendet – oder aber durch die im Prozessor integrierte PLL vervielfacht werden. Da ein Schwingquarz billiger als ein Oszillator ist, entschieden wir uns, einen Quarz einzusetzen. Da wir auf die maximale Frequenz des Prozessors von 72 MHz kommen wollten, wir einen 8MHz Quarz auf Lager hatten und sich mit diesem durch die PLL genau die gewünschte Frequenz erreichen ließ, entschieden wir uns für den 8.0MHz Quarz.
Auf die Konfiguration der PLL kommen wir gleich noch zurück.

5.4.4 *Clock Oszillator*

Zusätzlich planten wir für unsere Print noch einen Uhrenquarz mit 32.768 kHz ein, da der Prozessor über eine in Hardware integrierte Uhr mit Kalenderfunktion verfügt. So ist es möglich, die Uhr auch in Stromsparzuständen des Prozessors zu verwenden. Diese Uhr ist zwar hardwaremäßig vorgesehen und wurde softwaremäßig getestet, ist aber lediglich als Zusatzfeature auf dem Demonstrationsboard vorgesehen.

5.4.5 *Konfiguration*

Wenden wir uns nun der Konfiguration der Main Clock zu. Um von den 8.0 MHz des Quarzes auf die Maximalfrequenz des Prozessors zu kommen, wird die interne PLL verwendet.

Bei der internen PLL wird die Eingangsfrequenz zuerst durch (N +1) (Wert zwischen 1 und 256) geteilt. Dadurch ergeben sich mehr Einstellmöglichkeiten für die Outputfrequenz. Anschließend wird diese Frequenz mit 2* (M + 1) (Wert zwischen 1 und 32768) multipliziert. Das resultierende Signal (FCCO) muss eine Frequenz zwischen 275 und 550MHz besitzen. Anschließend wird diese Frequenz durch CCLKSEL + 1 dividiert um die eigentliche Frequenz für den Prozessorkern zu erhalten. CCLKSEL hat einen Wertebereich zwischen 0 und 255, wobei nur ungerade Werte für dieses Register verwendet werden dürfen. Diese resultierende Frequenz wird auch für die Peripherie des Prozessors genutzt. Wobei hier dann noch Teiler von 1, 2, 4 und 8 einstellbar sind. (Ausnahme CAN, hier gibt es statt 8 den Teiler 6)
Um also von unseren 8 MHz auf 72 MHz zu kommen haben wir folgende Registerwerte benutzt:
N=0; M=17; CCLKSEL=3;
FCCO = (2 × (M + 1) × FIN) / (N + 1)
Fcco=(2*18*8) = 288 MHz

Die Bedingung, dass Fcco zwischen 275 und 550MHz ist, ist also erfüllt. Durch Dividieren dieser Frequenz durch (CCLKSEL + 1) erhalten wir für CCLK die gewünschte Frequenz von 72 MHz.

5.4.6 *Konfigurationsablauf*

Wie schon zuvor ist für das Setzen dieser Register ein Codesegment im Startupcode vorgesehen. Da es wichtig ist, dass die PLL nicht durch einen Softwarefehler irrtümlich verstellt werden kann – dies könnte z.B. den Watchdog außer Betrieb setzen – ist ein spezieller Ablauf notwendig, um die PLL zu konfigurieren.
Um ein PLL Register zu schreiben, muss zuerst der neue Wert auf das jeweilige Holding Register, und direkt anschließend ohne Unterbrechung die Sequenz 0xaa und 0x55 auf das PLLFEED Register geschrieben werden. Weiters muss zuerst die PLL gestoppt werden, um die Werte von M und N auf das PLLCFG Register zu schreiben.

Der korrekte Ablauf dafür sieht also wie folgt aus:

Zuerst muss der Prozessor von der PLL getrennt und direkt mit dem Clock versorgt werden. Der Grund dafür ist – wie vorher schon erwähnt – dass es durch den Bootloader (beim Debuggen) dazu kommen kann, dass dieser die PLL bereits konfiguriert hat.

Dies geschieht, indem auf das PLLCON der Wert 1 gefolgt von einer Feed Sequenz geschrieben wird. Damit ist die PLL zwar immer noch aktiviert, aber nicht mehr mit dem Prozessorkern verbunden.
Anschließend kann die PLL gestoppt werden. Dies geschieht dadurch, dass der Wert 0 auf das PLLCON Register geschrieben wird – wiederum gefolgt durch eine Feed Sequenz.

Damit ist die PLL gestoppt und kann nun konfiguriert werden. Als Erstes muss hierfür der Main Oszillator überhaupt erst aktiviert werden. Dies geschieht, indem das OSCEN Bit im SCS Register gesetzt wird. Anschließend kann dieser als Eingang für die PLL gewählt werden. Dazu muss der Wert 0x01 auf das CLKSRCSEL Register geschrieben werden.
Als nächstes können die Werte für M und N auf das PLLCFG Register schreiben. Hier ist wieder eine Feed Sequenz nötig.

Damit ist die PLL bereits konfiguriert und wir können wieder beginnen, diese zu starten. Dazu müssen wir wieder den Wert 1 auf das PLLCON Register schreiben. Natürlich darf auch hier eine Feed Sequenz nicht fehlen. Bis die PLL eingerastet ist, können wir in der Zwischenzeit das CCLKSEL Register mit unserem Teilerfaktor setzen.
Nun müssen wir warten bis die PLL wirklich eingerastet ist. Dies geschieht indem wir in einer Schleife das Bit PLOCK aus dem Register PLLSTAT prüfen. Sobald das Bit logisch 1 ist, ist die PLL eingerastet und wir können in der Konfiguration fortfahren.
Der letzte Schritt in der Konfiguration ist nun, dass die PLL wieder als Clock für den Prozessor verbunden wird. Dazu müssen wir den Wert 0x03 auf das PLLCON, gefolgt von der letzten Feed Sequenz, schreiben. Damit ist die PLL wieder als Clock Source für den Prozessor definiert.
Wir müssen nun nur noch warten bis dies auch tatsächlich geschehen ist. Dazu können wir wieder ein Bit im PLLSTAT abfragen, nämlich das PLLC Bit. Ist auch dieses Bit gesetzt, so ist die Clock Konfigurationssequenz abgeschlossen.

5.5 *Verwendung von Prozessorpins*

5.5.1 *Allgemein*

Um einen Prozessorpin als einfachen Ein- oder Ausgang, oder aber als eine spezielle Funktion für eine interne Peripherie verwenden zu können, muss der jeweilige Pin zuerst konfiguriert werden. Auf den ersten Blick mögen die folgenden Einstellung den Anschein erwecken, sehr zahlreich zu sein. Dies ist aber notwendig, da der Prozessor über sehr viel interne Peripherie verfügt, und trotzdem sehr individuell konfigurierbar ist. Dadurch lassen sich sehr weit gefächerte Kombinationsmöglichkeiten – was die Peripherie betrifft – erreichen und sich der Prozessor so für viele Anwendungsgebiete einsetzen.

Prinzipiell lassen sich alle Ports des LPC2478 (Port 0 bis Port 4) über die so genannten Fast Register ansprechen. Diese sind auf den lokalen ARM Bus gemapped und so direkt mit dem Prozessorkern verbunden. Dadurch ist die schnellstmögliche Zugriffszeit garantiert. Weiters ist dadurch auch ein zusätzliches Feature, ein maskierter Zugriff auf den Port, möglich. Zusätzlich sind die Ports P0 und P1 alternativ über Register steuerbar, die über den peripheren Bus angesprochen werden müssen. Diese Register haben historische Gründe um die Softwarekompatibilität mit älteren Prozessoren sicherzustellen. Wir wollen uns deshalb hier auf die Fast Register beschränken.

Um nun einen Prozessorpin verwenden zu können, sind folgende Einstellungen notwendig:

5.5.2 *Betroffene Register*

5.5.2.1 *Funktion festlegen*

Als Erstes aktivieren wir die Fast Register für Port 0 und Port 1. Dazu müssen wir im System Controls and Status Register (SCS) das GPIOM Bit setzen.

```
SCS |= (1<<GPIOM);
```

Nun können wir alle Ports gleich ansprechen. Um nun einen Pin zu verwenden, müssen wir folgende Register konfigurieren:
PINSEL0 bis PINSEL9 – Diese Register steuern den Verwendungszweck der einzelnen Pins. Jeweils zwei Bit gehören zu einem Pin. Also PINSEL0 gehört zu P0.0 – P0.15, PINSEL1 zu P0.16 – P0.31, usw.
00b stellt hier immer den Resetwert dar. Das ist für gewöhnlich die Konfiguration als einfacher digitaler Ein- oder Ausgang.

PINMODE0 bis PINMODE9 – Diese Register sind für interne Pull up/down Widerstände zuständig. Die Zuordnung zu den jeweiligen Pins ist ident mit denen der PINSEL Register. Ebenfalls ist hier der Defaultwert 0x00. Hier die Bedeutungen der vier Einstellmöglichkeiten:

Tabelle 5: Übersicht Pull up/down

00	Pull up Widerstand aktiviert (default)
01	Nicht verwendet
10	Weder Pull up noch Pull down aktiviert
11	Pull down ist aktiviert

5.5.2.2 *Digital I/O*

Ist nun ein Port als digitaler Ein/Ausgang definiert, können wir über folgende Register direkt auf die Werte zugreifen.
Als erstes können wir konfigurieren, ob ein Portpin ein Ein- oder Ausgang sein soll. Dies geschieht über das FIOxDIR Register. Bei allen nun aufgeführten Registern gilt, dass mit x der jeweilige Port gemeint ist (also 0 bis 4). Weiters sind alle Register 32Bit breit. Damit kann jeder Portpin einzeln gesteuert werden.
Mit FIOxPIN kann der jeweilig tatsächliche Zustand eines Portpins eingelesen werden.
FIOxSET dient dazu einzelne Pins zu setzen. Setzt man ein Bit in diesem Register, so wird der korrespondierende Pin gesetzt. Schreibt man eine Null auf dieses Register, hat das keinen Effekt.
Ähnlich dazu gibt es natürlich auch ein FIOxCLR Register. Mit Hilfe dessen kann man Portpins wieder auf logisch low setzen. Hier gilt das selbe wie für FIOxSET – eine Null auf dieses Register zu schreiben hat keine Wirkung.

Dann gibt es noch das FIOxMASK Register. Damit lassen sich Zugriffe auf das Portregister maskieren, und die ausmaskierten Bits werden nicht beeinflusst. Das bedeutet, setzt man beispielsweise das erste Bit dieses Registers, also schreibt man 0x0000 0001 auf das FIOxMASK Register, so ist das erste Bit beim Auslesen von FIOxPIN immer Null und Schreibzugriffe auf das erste Bit von FIOxSET oder FIOxCLR haben keine Wirkung.

5.6 *External Memory Controller*

5.6.1 *Allgemein*

Da unser Prozessor einen integrierten Displaycontroller besitzt und wir damit ein Display mit wenigstens 800x480x16 Bit betreiben wollen, mussten wir schon beim Design unseres Demoboards einen externen Arbeitsspeicher vorsehen. Zur Anbindung an unseren Prozessor sind eigene Pins dafür vorgesehen, über die ein interner Speichercontroller die notwendigen Timings, Refreshs, etc. generieren kann, sodass aus Sicht der Software der Inhalt direkt angesprochen werden kann. Der Speichercontroller verbirgt den Zugriff auf das externe RAM, sodass es im Programm selbst – von der Syntax – keinen Unterschied macht, ob eine Variable im internen oder externen RAM liegt.
Prinzipiell würde unser Prozessor mit integriertem Speichercontroller sowohl SRAM als auch SDRAM unterstützen. Wir haben uns für SDRAM entschieden, da diese größer und billiger sind, und außerdem weiter verbreitet sind. Außerdem entschieden wir uns das RAM mit 32bit Datenbreite an den Prozessor anzubinden. Dadurch lässt sich ein höherer Datendurchsatz erreichen. Das ist vor allem wichtig, wenn gleichzeitig das Display aus dem RAM mit Daten versorgt wird und auch der Prozessor auf Speicherinhalte zugreifen muss.
Im Prinzip stellt uns auch für diese Funktion das Entwicklungsstudio einen Programmcode zur Verfügung, der die Register konfiguriert. Allerdings müssen wir diesen etwas anpassen, um mit unserer RAM-Konfiguration arbeiten zu können.

5.6.2 *Pinkonfiguration*

Als erstes müssen wir, um das externe RAM konfigurieren zu können, die Prozessorpins, die mit dem RAM verbunden sind, konfigurieren, damit diese vom Speichercontroller verwendet werden können.
Das PORT3 stellt zur Gänze den 32 Bit breiten Datenbus zwischen Prozessor und Arbeitsspeicher dar. Daher müssen wir laut Datenblatt für alle Pins den Wert 01b auf das jeweilige PINSEL Register schreiben. Dazu schreiben wir also den Wert 0x5555 5555 auf das Register PINSEL6 und PINSEL7. Weiters deaktivieren wir die Pull up/down Widerstände dieser Portpins. Dies geschieht, indem wir auf das PINMODE6 und PINMODE7 Register den Wert 0xAAAA AAAA schreiben. Damit haben wir die Pins für den Datenbus erstmal konfiguriert.
Nun konfigurieren wir P2.24 als Clockout für die SDRAM Bausteine, P2.16 als Coloumn Select, P2.17 als Row Select, P2.20 als Dynamic Chip Select und setzen zugleich P2.28 – P2.31 als Data Mask Leitung für den RAM Zugriff. Normalerweise müssten wir dazu den aktuellen Zustand aus dem PINSEL5 auslesen, die von uns benötigten Werte ausmaskieren und setzen, und anschließend wieder zurückschreiben. Da wir zum aktuellen Zeitpunkt noch keinen anderen Pin verwendet haben, können wir die von uns unbenutzten Bit auch einfacher auf den Defaultwert (00b) setzen. Damit können wir uns das Auslesen und Ausmaskieren ersparen.
Es ergibt sich für uns der Registerwert 0x5501 0115, den wir direkt auf das PINSEL5 schreiben können. Weiter deaktivieren wir die Pull up/down Widerstände für die von uns verwendeten Pins indem wir den Wert 0xAA02 022A auf das PINMODE5 Register schreiben.

Tabelle 6: EMC Port 2

Betreffender Pin	P2.31		P2.30		P2.29		P2.28		P2.27		P2.26		P2.25		P2.24	
Bit im PINSEL 5	Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	Bit 23	Bit 22	Bit 21	Bit 20	Bit 19	Bit 18	Bit 17	Bit 16
Wert	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	1
Betreffender Pin	P2.23		P2.22		P2.21		P2.20		P2.19		P2.18		P2.17		P2.16	
Bit im PINSEL 5	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Wert	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1

Nun zu den PINs, die für die Adressleitungen zuständig sind. Einmal vorweg zur Organisation unseres RAMs. Wir haben uns ja entschieden, auf unserem Demonstrationsboard – falls es für spätere Anwendung notwendig sein sollte – 512 MBit RAM zu installieren. Weiters besitzen die von uns eingesetzten RAMs 13 Reihen, 9 Spalten und das Ganze zu vier Bänken. Dadurch, und weil unsere RAM Bausteine zu 32bit Datenbussbreite verschaltet sind, ergibt sich die Gesamtspeichergröße von 512 MBit. (2^9 * 2^13 * 4 Bänke * 32 Bit = 512Mbit)
Die von uns also benötigten Adressleitungen und Bank Select Leitungen befinden sich auf dem PORT4. P4.0 bis P4.12 stellen die Adressleitungen dar, P4.13 und P4.14 die Bank Select Leitungen. P4.15 wird von uns nicht benötigt. Da wir diesen Pin nach einem Reset noch nicht verwendet haben, können wir uns – wie schon zuvor – das Ausmaskieren erneut ersparen und wieder direkt den Resetwert des Pins auf das PINSEL Register schreiben. Daraus ergibt sich für das PINSEL8 Register der Wert 0x1555 5555. Weiters deaktivieren wir wieder die Pull up/down Widerstände der verwendeten Pins indem wir 0x2AAA AAAA auf das PINMODE8 Register schreiben.

Tabelle 7: EMC Port 4

Betreffender Pin	P4.15		P4.14		P4.13		P4.12		P4.11		P4.10		P4.9		P4.8	
Bit im PINSEL 8	Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	Bit 23	Bit 22	Bit 21	Bit 20	Bit 19	Bit 18	Bit 17	Bit 16
Wert	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Betreffender Pin	P4.7		P4.6		P4.5		P4.4		P4.3		P4.2		P4.1		P4.0	
Bit im PINSEL 8	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Wert	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Nun fehlt uns nur noch der Pin für das Write Enable Signal. Dieser befindet sich ebenfalls am PORT4, nämlich am P4.25. Hier gilt wieder dasselbe wie zuvor. Da wir die anderen P4.X Pins, die vom PINSEL9 Register betroffen sind noch nicht verwendet haben, können wir uns wieder das Ausmaskieren ersparen. Somit können wir einfach den Wert 0x0004 0000 auf das PINSEL9 Register schreiben, um den Pin

P4.25 als Write Enable Signal zu definieren und den Wert 0x0008 0000 auf das PINMODE Register, um die Pull up/down Widerstände zu deaktivieren.

Damit ist die Pinkonfiguration für unser externes RAM abgeschlossen.
Das nachfolgende Codesegment zeigt das Programmieren dieser Register in Assembler:

```
ldr r0, =PCB_BASE

ldr r1, =0x55555555;
str r1, [r0, #PINSEL6_OFFSET]
str r1, [r0, #PINSEL7_OFFSET]
ldr r1, =0xAAAAAAAA;
str r1, [r0, #PINMODE6_OFFSET]
str r1, [r0, #PINMODE7_OFFSET]

ldr r1, =0x0x55010115
str r1, [r0, #PINSEL5_OFFSET]
ldr r1, =0x0xAA02022A
str r1, [r0, #PIMODE5_OFFSET]

ldr r1, =0x15555555;
str r1, [r0, #PINSEL8_OFFSET]
ldr r1, =0x2AAAAAAA;
str r1, [r0, #PINMODE8_OFFSET]

ldr r1, =0x00040000;
str r1, [r0, #PINSEL9_OFFSET]
ldr r1, =0x00080000;
str r1, [r0, #PINMODE9_OFFSET]
```

5.6.3 *Konfiguration des Speichercontrollers*

Nachdem die Ports nun konfiguriert sind, müssen wir als Erstes den Speichercontroller einschalten. Dazu müssen wir im PCONP Register das Bit 11 (PCEMC) setzen. Damit wird der Speichercontroller mit Strom versorgt. Im Nachfolgenden der kurze Assembler Ausschnitt, der das Setzen dieses Bits zeigt. Hier sieht man, dass zuerst der Wert des Registers ausgelesen werden muss, dass das Bit mit einer Oder Verknüpfung gesetzt wird, und anschließend der Wert wieder zurück geschrieben werden muss. Dieses Prozedere konnten wir uns bei den

PINSEL Register ersparen, da diese zu dem verwendeten Zeitpunkt noch Ihren Resetwert hatten.

```
ldr r0, =SCB_BASE
ldr r1, [r0, #PCONP_OFFSET]     //Hier wird der Wert von PCONP in das Register R1 geladen
orr r1, r1, #PCONP_PCEMC                //zusätzliches Setzen des PCEMC Bits
str r1, [r0, #PCONP_OFFSET]     //zurück schreiben des neuen Wertes auf das PCONP Register
```

Als Nächstes galt es, die Konfigurationsregister des Speichercontrollers anzupassen. Prinzipiell war – wie schon zuvor erwähnt – das Setzen der Register bereits im Startupcode vorgesehen. Trotzdem mussten wir natürlich die Werte für die Register erst an unser Demonstrationsboard anpassen. Während der Entwicklung unseres Demoboardes wurden uns gleich zwei Mal die von uns vorgesehenen Speicherbausteine abgekündigt. Das erste Mal von Samsung, das zweite Mal von Winbond. Die nachfolgenden Timingwerte für den Speichercontroller sind nun für keinen speziellen Hersteller optimiert, sondern sollten die Timingspezifikation der meisten Hersteller erfüllen. Weiters sind diese Timingwerte mit Speicherbausteinen von Winbond, Micron und die von uns favorisierten SDRAMs von ESTM getestet. In einer Serienproduktion, oder wenn die Performance der Plattform absolut ausgereizt wird, kann es unter Umständen nötig sein, diesen Wert genau auf die verwendeten Speicherbausteine anzupassen.
Die nachfolgende Übersicht soll einen Überblick auf die von uns gewählten Werte liefern. Die Initialisierungsprozedur kann im Datenblatt und in der Applikation Note von NXP nachgelesen werden.

Als erstes der Konfigurationswert für EMC_DYN_CFG0:
Hier haben wir 0x00085680 errechnet: 32Bit external Bus, low Power SDRAM, 13 Reihen, 9 Spalten, 4 Bänke, Buffer enabled.
Für RAS und CAS haben wir jeweils drei Clock Zyklen gewählt. Weiters wählten wir für:
Dynamic Memory Refresh Timer: 560 CCLK
Dynamic Memory Percentage Command Period: 3 CCLK
Dynamic Memory Active to Precharge Command Period: 5 CCLK
Dynamic Memory Self-refresh Exit Time register: 6 CCLK
Dynamic Memory Last Data Out to Active Time: 2 CCLK

Dynamic Memory Data-in to Active Command Time: 5 CCLK
Dynamic Memory Write Recovery Time register: 8 CCLK
Dynamic Memory Active to Active Command Period register: 6 CCLK
Dynamic Memory Auto-refresh Period register: 6 CCLK
Dynamic Memory Exit Self-refresh register: 6 CCLK
Dynamic Memory Active Bank A to Active Bank B Time register: 3 CCLK
Dynamic Memory Load Mode register to Active Command Time: 4 CCLK

5.6.4 *Verwendung*

Nach dem Setzen der Register und dem Senden der Initialisierungssequenz ist das externe RAM direkt aus dem Programmcode heraus ansprechbar. Der Adressbereich unseres externen RAMs geht dabei von 0xA000 0000 bis 0xA400 0000. In diesem Bereich können Variablen definiert werden, auf die – genauso wie auf interne Variablen – zugegriffen werden kann. Es sollte bei der Platzierung von Variablen allerdings immer darauf geachtet werden, dass „kleine“ Variablen, sowie Variablen, auf die häufig zugegriffen wird, im internen RAM gespeichert werden, da hier die Zugriffszeit schneller ist. Sehr große Speicherbereiche, oder solche, auf die der Prozessorkern gar nicht (Speicherbereiche werden nur per DMA geschrieben und gelesen) oder nur sehr selten zugreift, können im externen RAM platziert werden.

5.7 *Speicherkartenanbindung*

5.7.1 *Allgemein*

Um unser Demonstrationsboard mit Daten zu versorgen, haben wir beim Entwurf eine SD-Karte eingeplant. Diese würde sich zum einen über SPI ansprechen lassen oder aber, wie bei uns, über einen in unserem Prozessor integrierten SD/MMC Card Controller. Dieser Controller erzeugt das Clocksignal für die SD Karte, bietet schnelle Datenübertragungsraten und generiert alle notwendigen Interrupts, um komfortabel mit der Speicherkarte arbeiten zu können.
Prinzipiell wäre es bei uns nicht notwendig gewesen, ein Filesystem auf der SD-Karte zu haben. Wir könnten einfach durch direktes Ansprechen von Sektoren, Daten von der Speicherkarte abfragen. Jedoch würde dies vor allem das Aufspielen von Daten auf die Karte komplizierter gestalten. Da wir mit unserem Demonstrations-board eine Plattform schaffen wollen, die später einmal individuell konfigurierbar ist, entschlossen wir uns, auf der Speicherkarte ein Filesystem zu verwenden. Dadurch lassen sich Dateien mit einem herkömmlichen PC mit Kartenlesegerät direkt auf die Speicherkarte übertragen.

5.7.2 *Filesystem*

Für die Speicherkartenanbindung haben wir in unserer Plattform ein Filesystem implementiert. Wir entwickelten dieses jedoch nicht selbst, sondern implementierten FatFs von ChaN. Dieses Filesystem hat einen inzwischen sehr weit fortgeschrittenen Funktionsumfang und es gibt von NXP eine eigene Application Note[2], um dieses Filesystem auf deren Hardware zu portieren. Bei diesem Filesystem handelt es sich um ein speziell für Embedded Systeme programmiertes Filesystem, das sowohl FAT12, FAT16 als auch FAT32 unterstützt. Dadurch ist dieses System sehr Ressourcen sparend programmiert, was speziell unserer Software Updatefunktion zu Gute kommt. Ein weiterer wichtiger Pluspunkt dieses Filesystems ist, dass dieses frei verfügbar ist. Es darf sowohl für Ausbildungszecke, für private sowie auch für kommerzielle Projekte frei eingesetzt werden.

[2] NXP - AN10916

Dieses Kapitel haben wir deshalb hier eingeschoben, da dieses Filesystem im nachfolgenden Kapitel gleich zum ersten Mal verwendet wird. Im diesem beschäftigen wir uns noch nicht mit der eigentlichen Applikation, sondern mit der von uns implementierten Möglichkeit, das Benutzerprogramm direkt aus der Anwendung heraus neu zu aktualisieren (IAP – In Application Programming). Das bedeutet, dass wir im laufenden Programm den Inhalt des Flashspeicher löschen und mit neuen Programmdaten beschreiben können.

5.8 *Softwareupdate – IAP – In Application Programming*

5.8.1 *Allgemein*

Die bisher aufgelistete Software, das Definieren der Einsprungvektoren, das Einstellen des Prozessortaktes, das Konfigurieren des MAM und des Speichercontrollers gehören eigentlich noch nicht zum eigentlichen Programm, sondern dienen vielmehr dazu, die Grundkonfiguration des Prozessors herzustellen. Nach dieser Grundkonfiguration kann im Normalfall schon das eigentliche Programm aufgerufen werden. In den meisten Fällen die void main(void) Funktion. Wir haben aber in unserer Software noch einen Zwischenschritt eingebaut. Eine Art Erweiterung des Startup-Codes. Um unseren Prozessor auch ohne Debuggtool oder sonstige direkte Schnittstelle mit einem Computer programmieren zu können, haben wir ein zusätzliches Feature eingebaut. Eine „In Application Programming“ (IAP) Funktion. Damit kann man später in der Entwicklung bei Feldversuchen oder Dergleichen an der Software Änderungen vornehmen, ohne das Gerät zerlegen zu müssen, um an die Programmierpins zu kommen. Weiters bietet diese Funktion sogar Endkunden/-innen später die Möglichkeit, Softwareupdates selbst in das Gerät einzuspielen.
Dazu muss lediglich die Datei mit der Firmware auf der SD-Karte unter einem bestimmten Namen abgelegt werden und bei Einschalten des Gerätes ein Portpin auf Masse gezogen werden. Danach prüft das System, ob sich eine Firmwaredatei auf der SD-Karte befindet. Ist dies der Fall, programmiert die Firmware die neue Software in den Flashspeicher des Prozessor und resettet diesen anschließend. Ist keine Firmwaredatei auf der Speicherkarte vorhanden, wird das normale Programm gestartet.
Wie dieser Mechanismus im Detail funktioniert werden wir nun erörtern.

5.8.2 *Grundlagen*

Einmal vorweg zu Erinnerung: Das Flash des Prozessors verfügt über 28 Sektoren. Die ersten 8 Sektoren haben 4kB, die nächsten 14 haben jeweils 32 kB, und die

letzten 6 Sektoren haben wieder lediglich 4kB. Das ergibt eine Gesamtgröße von 504kB.
Nach diesen 28 Sektoren gibt es noch den prozessoreigenen Boot Block. Dieser dient dazu, ISP Funktionen zur Verfügung zu stellen, beim Programmstart zu prüfen, ob über die serielle Schnittstelle programmiert werden soll, etc. Weiters können auch über spezielle Adressen in genau diesem Adressbereich Funktionen aufgerufen werden, mit denen es möglich ist, einzelne Flashsektoren aus dem laufenden Betrieb zu bearbeiten. Dieser Block kann nicht gelöscht oder überschrieben werden.

5.8.3 *Standardablauf*

Der übliche und empfohlene Programmablauf bei einer Firmwareaktualisierung sieht wie folgt aus. Zuerst wird ein (manchmal sogar ausgelagerter) Programmteil zum Aktualisieren der Firmware ins RAM kopiert und die Befehlsausführung des Prozessors an diese Adresse verlagert. Das bedeutet, der Programmcounter zeigt nun auf eine Adresse im RAM. Auch Interrupts müssen entweder abgeschalten, oder die Einsprungadressen ins RAM gemapped werden. Sobald kein Zugriff auf den Flashspeicher mehr stattfindet, wird der gesamte Inhalt des Flashs gelöscht und neu programmiert. Diese Updatevariante hat den Vorteil, dass der gesamte Programmspeicher neu programmiert wird, und die im Flash implementierte Updatefunktion nur sehr wenig Programmspeicher benötigt (Das eigentliche Updateprogramm könnte man ja schon wieder auf die Speicherkarte auslagern). Weiters entstehen auch bei der Programmerstellung keine Lücken im Flash, da der Programmcode fortlaufend über die Sektoren hinweg erstellt werden kann. Einen ganz gravierenden Nachteil hat diese Methode der Softwareupdatefunktion jedoch. Tritt während des Softwareupdates ein Fehler auf (z.B. ein Einbruch der Versorgungsspannung), oder hat die neue Firmwaredatei einen Fehler, oder ist ganz einfach die neue Firmware fehlerhaft, so ist die Updatefunktion kaputt. Handelt es sich bei dem Update jetzt um ein Softwareupdate eines Gerätes direkt beim Endkunden/-in, so ist das Gerät jetzt unbrauchbar. Er/Sie hat keine Möglichkeit mehr, das Gerät zum Laufen zu bringen, ohne es an den Hersteller einzuschicken oder sich ein neues Gerät zu kaufen.
Um diese Problematik zu umgehen, haben wir uns für ein alternatives Updateverfahren entschieden, bei dem es (fast) nicht passieren kann, dass die

Softwareupdatefunktion zerstört wird. Eines dazu noch vorweg, auch diese Methode hat Nachteile – doch dazu etwas später.

5.8.4 *Die eingesetzte Update Funktion*

Der große Unterschied in unserem Systementwurf zu der von NXP vorgeschlagenen Variant ist, dass unsere Firmwareupdatefunktion sich ebenfalls im Flashspeicher befindet.
Das bedeutet, dass wir eigene Sektoren definiert haben, die für die Updatefunktion zuständig sind. Diese Sektoren werden bei einem Systemstart von den Routinen, die auf den Flashspeicher schreiben, nicht verändert, dadurch – und durch die passende Wahl dieser Sektoren – kann sichergestellt werden, dass die Updatefunktion immer erhalten bleibt, auch wenn bei einem Systemupdate ein Fehler auftritt oder die neue Firmware fehlerhaft ist.
Im vorherigen Kapitel (5.8.3) haben wir „fast“ geschrieben, als wir davon sprachen, dass die Updatefunktion nie zerstört werden kann. Dieses „fast“ kommt daher, dass wir uns entschieden haben, auch den Sektor 0 mitzuprogrammieren. Dadurch lassen sich mit einem Softwareupdate auch die Ziele der Einsprungvektoren einfach mitverändern und der übrige Speicherplatz von Sektor 0 lässt sich weiters als Programmspeicher nutzen.

Um die Gefahr einer Zerstörung der Softwareupdatefunktion zu minimieren, wird bei einem neuen Programm als Erstes überprüft, ob der Resetvektor wieder in die Softwareupdateroutine zeigt. Ist dies nicht der Fall, wird die neue Software nicht aufgespielt. Weiters wird der Sektor 0 als Letztes gelöscht und neu programmiert. Das bedeutet, dass sich zuerst alle anderen 27 Sektoren (die teilweise acht mal so groß sind wie Sektor 0) erfolgreich programmieren haben lassen, und erst dann der Sektor 0 programmiert wird. Ist zuvor ein Fehler aufgetreten, wird der Updatevorgang unterbrochen und die Updatefunktion bleibt erhalten.

Für die korrekte Funktion unserer Updatefunktion sind also zwei Sektoren entscheidend:
Sektor 0: Hier stehen die Einsprungvektoren. Auf dem Resetvektor muss ein Sprung in die Updatefunktion stehen. Dies wird bei einer neuen Firmware immer zuerst

überprüft. Nur wenn in der neuen Software am Resetvektor ein Sprung in die Updatefunktion steht, wird diese akzeptiert und in den Prozessor programmiert.
Sektor 21: Hier befindet sich das Filesystem, das notwendig ist, um das neue Programm von der SD-Karte und die für das Flashupdate notwendigen Funktionsaufrufe zu laden.

5.8.5 *Zusätzlich benötigter Speicherbedarf*

Zuerst einmal zum zusätzlichem Speicherbedarf, den unsere Updatefunktion im Flashspeicher benötigt. Prinzipiell ist nur ein Sektor belegt: Sektor 21 mit 32 kB. Das würde rechnerisch etwa 6,4% des gesamten Flash ausmachen. Wir haben hier aber absichtlich den Konjunktiv verwendet, da diese Rechnung nicht ganz stimmt.
Unsere Firmware Updatefunktion ist so entworfen, dass diese bei jedem Systemstart als Erstes gestartet wird. In dieser Updatefunktion wird überprüft, ob ein Systemupdate durchgeführt werden soll. Wenn dies nicht der Fall ist, startet die Updatefunktion das eigentliche Programm. Ist ein Update durchzuführen, wird das zuerst erledigt und danach ebenfalls das normale, neue Programm gestartet. Um ein Update aus programmtechnischer Sicht komfortabel durchführen zu können, benötigen wir schon einmal unser externes RAM. Wenn wir aber unser externes RAM benötigen, müssen wir zuerst den Systemtakt herstellen. Das bedeutet also, dass wir (fast) den gesamten Startup Code bereits in der Firmwareupdate Funktion benötigen. Daher können wir aber gleich den gesamten Startup Code in die Update Funktion packen und dafür im eigentlichen Programm weglassen. Bevor das eigentliche Programm gestartet wird, wird sowieso zuerst die Firmwareupdate-funktion gestartet und damit zuerst der Startupcode ausgeführt. Damit ist sichergestellt, dass immer zuerst der Startupcode ausgeführt wird, bevor der Einstieg in das eigentliche Programm stattfindet.
Schlussendlich aber reduziert sich der Speicherbedarf unseres eigentlichen Programms, wodurch der berechnete zusätzliche Speicherbedarf nicht mehr ganz stimmt.

5.8.6 *Ablauf*

Nun zum eigentlichen Ablauf beim Starten des Systems. Wie wir ja bereits wissen, startet unser Prozessor bei einem Reset oder Systemstart und prüft (mit dem

internen Bootlaoder) ob ein externer Request für das direkte Aktualisieren des Programmspeichers durch das auf Masse ziehen eines Pins vorliegt, sodass er auf neue Daten über den UART0 warten soll. Ist dies nicht der Fall, wird überprüft, ob die Summe der Einsprungvektoren plus der Prüfsumme null ergibt. Ist dies nicht erfüllt, so wartet der Prozessor wiederum auf neue Daten über die serielle Schnittstelle. Ist aber auch diese Bedingung erfüllt, springt der Prozessor an die Adresse 0x0000 0000 und beginnt mit der Codeausführung. An dieser Adresse (am Resetvektor) steht ein Sprung in den von uns reservierten Sektor. Dies ist der letzte Sektor im Flash mit 32 kB, und hat damit die Startadresse 0x0007 0000.
An dieser Startadresse befindet sich als erstes unser Startup Code. In diesem wird als Prozessorclock der Main-Oszillator mit prozessorinterner PLL konfiguriert, das MAM konfiguriert, und der EMC konfiguriert – also all jene Dinge die wir zuerst schon erörtert haben. Anschließend beginnt nun endlich die Ausführung unseres C – Programms zum eventuellen Flashupdate.
Als Erstes wird in dieser Funktion ein Prozessorpin als Eingang mit aktiviertem Pull-Up-Widerstand konfiguriert. Anschließend wird dieser Pin ausgewertet. Wird von dem Port Pin eine logische eins gelesen, so wurde dieser nicht mit einem Taster auf Masse gezogen und kein Softwareupdate angefordert. In diesem Fall wird ein Sprung an die Adresse 0x0000 0040 durchgeführt. Das ist die erste Adresse hinter den Einsprungvektoren und den dazugehörigen Konstanten. Somit muss das eigentliche Programm immer an der Adresse 0x0000 0040 beginnen.
Ist der Port Pin auf Masse gezogen – also wurde eine Null vom Pin Register gelesen, so wurde dieser Portpin extern auf Masse gezogen. Dies wird als Aufforderung interpretiert, ein Softwareupdate durchzuführen.
In diesem Fall wird die Speicherkarte initialisiert und im Hauptverzeichnis nach dem File „prog.bin“ gesucht. Dieses File stellt das neue Programm dar. Existiert diese Datei, wird das neue Programm in das externe RAM eingelesen – existiert diese Datei nicht, so wird das Update abgebrochen.
Wurde die Datei erfolgreich in das RAM geladen, so wird nun als Erstes überprüft, ob das neue Programm einen korrekten Resetvektor besitzt. Das bedeutet, dass auch nach dem Softwareupdate der Prozessor weiterhin nach einem Reset automatisch in die Updatefunktion springt. Dazu werden die ersten vier eingelesen Byte überprüft, ob hier der richtige Load Befehl steht, und anschließend, ob an der Adresse 0x0000 0020 der Wert 0x0007 0000 steht. Damit ist sichergestellt, dass auch mit

diesem Programm der Prozessor nach dem Update an die Adresse 0x0007 0000 springen und sich dort die Updatefunktion starten wird.
Ist nun auch dieses Kriterium erfüllt, kann mit dem eigentlichen Update begonnen werden.

5.8.7 *Update Vorgang*

Für den Update Vorgang wird zuerst der VIC deaktiviert, sodass keine Interrupts mehr ausgelöst werden. Anschließend werden von hinten (also mit der höchsten Flashadresse) beginnend die neuen Daten ins interne Flash kopiert. Dazu werden die Funktionen aus dem internen Bootloader verwendet. Um diese Funktionen zu nutzen, existiert eine Funktion mit einer definierten Einstiegsadresse 0x7fff fff1, die weiters zwei Pointer auf unsigned long Arrays als Übergabeparameter erwartet. NXP zeigt in einer Application Note zu IAP wie man diese Funktion definiert:

```
typedef void (*IAP)(unsigned long [],unsigned long[]);
IAP iap_entry;
```
[3]

Anschließend lässt sich iap_entry wie eine normale C-Funktion mit zwei Übergabeparametern aufrufen. Dazu haben wir noch, wie von NXP vorgegeben, zwei Arrays definiert, die zur Übergabe der Befehle und der Returnwerte dienen:

```
unsigned long command[5];
unsigned long result[2];
```

Folgende Befehle die von iap_entry unterstützt werden sind für uns von Bedeutung:

Tabelle 8: IAP Befehle

Befehl	Befehlscode
Prepare sector(s) for write operation	50
Copy RAM to Flash	51
Erase sector(s)	52

[3] Auszug aus AN-10256 von NXP

Um einen Sektor schreiben zu können, muss dieser zuerst gelöscht werden. Weiters muss vor jeder Copy RAM to Flash sowie Erase sector Zugriff zuerst ein Prepare Setor Befehl ausgeführt werden. Das eigentliche Schreiben der neuen Daten ins Flash sieht wie folgt aus:
Zuerst werden 21 Flags gesetzt. Diese kennzeichnen, dass keiner der 21 Sektoren bereits gelöscht wurde.
Anschließend werden in einer Schleife jeweils 512 Byte aus dem externen RAM in einen Buffer ins interne RAM kopiert. Die Copy RAM to Flash kann nur Blöcke definierter Größe kopieren. Wir haben hier 512 Byte gewählt. Anschließend wird berechnet in welchem Sektor sich der Block befindet. Dies geschieht durch die Funktion Sektor.

```
unsigned char sektor (unsigned int addr)
{
  if(addr<0x8000)              //einer der ersten 4kB Sektoren
    return (addr/0x1000);
  if(addr<0x78000)             //ein 32kB Sektor
    return ((addr/0x8000)+7);
  if(addr<0x7e000)             //einer der letzten Sektoren
    return ((addr/0x1000)-98);

  return 21;                   //Sektor 21 ist der Bootloader
}
```

Kommt heraus, dass es sich bei der Adresse für die Daten um den Sektor 21 handelt – das entspricht unserem Bootloader – so wird dieser Datenblock übersprungen. Somit ist sichergestellt, dass unser Bootloader nicht überschrieben wird.
Handelt es sich nicht um unseren Bootloadersektor wird danach anhand des Flags geprüft, ob der Sektor bereits gelöscht wurde. Ist dies nicht der Fall – sprich das ist der erste 512 Byte Block – wird nun der Sektor gelöscht.
Anschließend wird der 512 Byte große Datenblock mit der Copy RAM to Flash Funktion in das Flash programmiert.
Danach beginnt die Schleife von vorne, bis der erste Sektor, Sektor 0 geschrieben ist. Als Letztes werden also die neuen Einsprungvektoren programmiert.

Nun zu den eigentlichen Funktionen:

Das Löschen eines Sektors sieht wie folgt aus:

//Zuerst eine Prüfung ob es sich bei dem Sektor nicht um die Stelle handelt an der unser Bootloader steht. Auf „i“ steht immer die Startadresse eines 512 Byte blocks der programmiert werden soll.

```
if(sektor(i)==21)
    continue;

//Anschließend das Vorbereiten des Sektors
    command[0]=50;                  //Befehlscode zum Vorbereiten
    command[1]=sektor(i);           //Sartsektor der Vorbereitet werden soll
    command[2]=sektor(i);           //Endsektor der Vorbereitet warden soll
    iap_entry(command,result);      //Aufruf der eigentlichen Routine
    if(result[0]!=0)                //Wenn dieser Wert ungleich Null ist,
    {  return;                      //so ist ein Fehler aufgetreten und das
    }                               //Update wird abgebrochen

//Hier findet das eigentliche Löschen statt
    command[0]=52;                  //Befehlscode zum Löschen
    command[1]=sektor(i);           //Startsektor
    command[2]=sektor(i);           //Endsektor (es könnten auch mehrere Sektoren
                                    //auf einmal gelöscht werden
    command[3]=CCLK;                //Taktrate des Prozessors in kHz
    iap_entry(command,result);      //Löschen wird durchgeführt
     if(result[0]!=0)               //Prüfung ob erfolgreich
    {  return;
    }
```

Ähnlich sieht auch das Schreiben eines Sektors aus. Dazu muss ebenfalls vor jedem Schreiben die Prepare Funktion aufgerufen werden (Auch wenn auf diesen Sektor schon für den vorherigen Schreibzugriff die Prepare Funktion aufgerufen wurde, muss diese erneut gestartet werden).
Hier nun die Schreibroutine für 512 Byte:

```
// Zuerst wieder das Vorbereiten des Sektors
command[0]=50;                      //Befehlscode zum Vorbereiten
command[1]=sektor(i);               //Startsektor, der vorbereitet werden soll
command[2]=sektor(i);               //Endsektor
iap_entry(command,result);          //Vorbereiten ausführen
if(result[0]!=0)                    //Returncode muss Null sein – sonst
    return;                         //ist ein Fehler aufgetreten und das
                                    //Update wird abgebrochen
```

```
command[0]=51;                          //Befehlscode für Copy RAM to Flash
command[1]=i;                           //Addresse im Flash
command[2]=(unsigned long)towrite;      //Zeiger auf die Daten im Ram die
                                        //programmiert werden sollen
command[3]=512;                         //Länge der Daten in Byte
command[4]=CCLK;                        //Taktrate des Prozessors in kHz
iap_entry(command,result);              //Aufruf der Programmierfunktion
if(result[0]!=0)                        //Funktion muss den Wert Null zurückgeben
    return;                             //sonst ist ein Fehler aufgetrete und
                                        //das Update wird abgebrochen
```

Am Ende eines Updates – egal ob dieses erfolgreich war oder nicht – wird der Prozessor neu gestartet. Dies geschieht mit Hilfe von:

```
asm("mov pc,#0");
```

Dadurch wird der Programmcounter auf Null gesetzt – also auf den Resetvektor – und der Prozessor beginnt von neuem das Programm zu starten.

5.8.8 *Update der Updatefunktion*

Wir möchten nun noch auf eine weitere Problematik unserer Version der Updatefunktion hinweisen. Bei uns ist, da wir den Startupcode sowohl in der Updatefunktion als auch im eigentlichen Programm benötigen – dieser im für die Updatefunktion reservierten Sektor platziert. Dies aber führt dazu, dass dieser nicht – oder nicht einfach – durch ein Softwareupdate aktualisiert werden kann.
Prinzipiell sind wir der Meinung, dass das keine Einschränkung darstellt. Der Grund dafür ist, dass sich in diesem Startupcode nur Grundeinstellungen befinden, für die wir keinen Grund wüssten, diese zu verändern. Unser Startupcode wurde von uns getestet und wir waren der Auffassung, dass dieser fehlerfrei funktioniert.
Sollte dennoch später einmal ein Softwareproblem mit dem Startupcode (oder mit unserer Updatefunktion) auftreten, so existiert auch mit unserer Updatefunktion die Möglichkeit, dieses zu beheben. Dafür müsste man lediglich ein Programm schreiben, das eine Updatefunktion – wie die von NXP vorgeschlagen – enthält, oder

aber den von uns reservierten Sektor direkt mit einer neuen Updatefunktion beschreibt. Diese Software würde sich ohne Weiteres mit unserer jetzt implementierten Updatefunktion einspielen lassen und würde diese danach updaten. Somit ist gezeigt, dass sich auch ein solches Problem später, auch wenn das Gerät bereits installiert ist, direkt beheben lassen würde.

5.8.9 *Initiale Programmierung*

Eine weitere Eigenschaft unserer Updatefunktion gibt es noch. Diesmal eine äußerst positive. Wenn später eine Serie von unserem Produkt gefertigt wird, muss vor der Auslieferung jedes einzelne Gerät programmiert werden. Sprich in jeden Prozessor muss unser Softwareprogramm eingespielt werden. Das kann entweder über eine Debugprobe (JTAG) oder über die serielle Schnittstelle geschehen. Diese Methode hätte vor allem den Vorteil, keine zusätzliche Hardware zu benötigen. Leider ist diese Programmiervariante nicht die schnellste. Hier kommt aber unsere Updatefunktion ins Spiel. Normalerweise müssten beim Programmieren des Prozessors bis zu 504kB an Daten über die serielle Schnittstelle übertragen werden. Dank unserer Updatefunktion reicht es aber, wenn nur diese über den UART übertragen wird. Dieses Programm alleine ist ja schon lauffähig, mit allem was dazu gehört (Initialisierung, Startupcode etc.). Ist die Updatefunktion übertragen, wird das Gerät einfach mit einer SD-Karte mit der gewünschten Firmware darauf eingeschaltet und das Gerät programmiert sich von selbst.
Dank dieses Mechanismus verringert sich der notwendige Programmieraufwand über die serielle Schnittstelle von 28 Sektoren auf lediglich 2 Sektoren. Nur der Sektor 0 mit den Einsprungvektoren und der Sektor 21 mit der eigentlichen Updatefunktion müssen programmiert werden. Das sind in Summe 36kB und entspricht lediglich 7,14% des gesamten Flash. Das entspricht also einer Zeitersparnis von über 90% beim Programmieren.
Auf die Speicherkarte müssen vor der Auslieferung so oder so die für das eigentliche Programm später notwendigen Daten kopiert werden. Dadurch entsteht beim Aufspielen der Firmwaredatei kein zusätzlicher Aufwand. Will man den Programmieraufwand von unserem Gerät noch weiter minimieren, wäre es noch möglich, im eigentlichen Programm eine Signatur an einer definierten Adresse im Flash zu hinterlegen. Die Updatefunktion könnte beim Starten diese Signatur

abfragen. Ist diese noch nicht im Programmspeicher vorhanden (der Prozessor ist noch nie mit dem eigentlichen Programm programmiert worden – ist also fabrikneu) wird das Update des eigentlichen Programms automatisch gestartet. Damit könnte in der Produktion das Aufspielen des eigentlichen Programms entfallen. Sobald das Gerät zum ersten Mal in Betrieb genommen wird, programmiert sich dieses unbemerkt und selbstständig. Somit wäre der Programmier- und damit der Zeitaufwand in der Produktion auf ein Minimum reduziert.
Außerdem entsteht durch diese äußerst schnelle Programmiervariante noch eine weitere Möglichkeit: Da das Programmieren über unsere Updatefunktion in rund zwei Sekunden abgeschlossen ist, ist es auch denkbar eine eigene Testversion zu programmieren. Diese wird bei der Produktion aufgespielt und kann dann beispielsweise zur Qualitätskontrolle verwendet werden. Ebenfalls könnte man auch in diesem Fall das Programmieren mit der für den Echtbetrieb gedachten Firmware dem Gerät selbst überlassen, sobald es in der Anlage zum ersten Mal in Betrieb genommen wird. Dafür würden wir den selben Trick wie zuvor beschrieben mit der Signatur verwendet können. Das Testprogramm müsste also ebenfalls eine gültige Signatur besitzen. Sobald das Testprogramm seinen Testlauf abgeschlossen hat, gibt es eine Bestätigung aus, dass das Gerät funktionstüchtig ist und zerstört dabei seine eigene Signatur. Damit wird das Gerät beim nächsten Systemstart automatisch wieder über die SD-Karte programmiert.

Wie wir in diesem Abschnitt gesehen haben, ist unsere Softwareupdatefunktion ein sehr nützliches Tool. Nicht nur in der Entwicklung (Softwareupdate bei Versuchen ohne Zerlegen des Gerätes), sondern auch in der Fertigung (schnelleres Programmieren) und im tatsächlichen Betrieb (Updatemöglichkeit, wenn Softwarefehler auftauchen oder zusätzliche Features zur Software hinzukommen). Diese Liste an Vorteilen rechtfertigt unserer Meinung nach den zusätzlichen Speicherbedarf im Flashspeicher.

5.8.10 *Projekt Gestaltung*

Nun möchten wir kurz erklären, wie wir diese Updatefunktion in unser Projekt eingebunden haben. Prinzipiell haben wir in unserem Entwicklungsstudio eine Arbeitsmappe mit zwei Projekten erstellt. Das eine Projekt ist die Updatefunktion, das

zweite Projekt ist das eigentliche Benutzerprogramm. Der Vorteil dieser Aufteilung ist, dass unser Compiler damit nicht weiß, dass diese Projekte zusammengehören. Damit können wir z.B. Variablen in unserer Updatefunktion nach Belieben anlegen, im Hauptprogramm werden diese Speicherbereiche einfach wiederverwendet. Der zweite große Vorteil ist, dass wir damit auch nur die Updatefunktion kompilieren können und damit sofort eine Datei haben, die wir direkt über die serielle Schnittstelle des Programms in den Prozessor programmieren können.
Allerdings haben wir uns mit dieser Aufteilung auch einen Nachteil eingefangen. So haben wir, wenn wir das Hauptprogramm kompilieren, nun keinen Startupcode dabei (dieser ist ja in der Updatefunktion beigepackt). Wenn wir also das Hauptprogramm direkt aus dem Entwicklungsstudio mit der Debugprobe auf unsere Zielhardware programmieren wollen, ist diese Software nicht lauf- oder debugfähig.
Zu diesem Zweck haben wir eine zweite Arbeitsmappe erstellt. Diese beinhaltet ein Projekt, welches alle Dateien und Einstellungen des Hauptprogrammes und zusätzlich einen Startupcode beinhaltet. Öffnen wir also unsere zweite Arbeitsmappe und kompilieren diese, so erhalten wir ein Programm, welches wir direkt auf die Zielplattform programmieren und auch debuggen können. Öffnen wir unsere erste Arbeitsmappe und kompilieren die Updatefunktion, so können wir diese ebenfalls sofort mit dem Debugger runterladen und debuggen, oder aber die Binärdaten über die serielle Schnittstelle in den Prozessor laden. Kompilieren wir in dieser Arbeitsmappe jedoch das Hauptprogramm so erhalten wir eine Datei, die wir auf die SD-Karte kopieren können und mit unserer Updatefunktion laden können. Die Funktion des Hauptprogrammes ist die selbe, ob wir diese mit der ersten Arbeitsmappe erstellt und über die Updatefunktion eingespielt haben, oder aber mit der zweiten Arbeitsmappe kompiliert und direkt eingespielt haben.

5.9 *LCD*

5.9.1 *Allgemein*

Nun zu den Funktionen, die wir für unser eigentliches Hauptprogramm benötigen. Als Erstes wollen wir unser Display in Betrieb nehmen. Um unser Display mit der physikalischen Auflösung von 800x480x16bit mit Daten versorgen zu können, haben wir ja schon zuvor erwähnt, dass wir den dafür notwendigen Ausgabepuffer im externen RAM platzieren müssen. Dieses haben wir ja schon im Startupcode konfiguriert und initialisiert und können dieses daher nun sofort verwenden.

5.9.2 *Display Puffer*

Jedes Pixel am Display benötigt 2Byte. 800 Spalten x 480 Zeilen ergeben 384.000 Bildpunkte. Das sind demnach 768.000 Byte, die für eine Anzeigeseite benötigt werden. Um nun mehrere Anzeigeseiten puffern zu können (damit diese nicht jedes Mal neu von der SD – Karte gelesen werden müssen) haben wir mehrere solcher Puffer erstellt. Die Anzahl der möglichen Puffer ist eigentlich lediglich durch die Größe des verfügbaren RAMs begrenzt. Wir haben uns hier für einen Puffer für 20 Bilder entschieden, da wir denken, dass dies für den Normalfall reichen sollte. Bei Bedarf kann dieser aber jederzeit erweitert werden. Außerdem benötigen wir einen zusätzlichen Puffer, um auf der gerade angezeigten Seite noch zusätzlich etwas einzuzeichnen bzw. dieses auf dem Hintergrund Eingezeichnete auch wieder entfernen zu können. Das bedeutet aber, dass der Puffer, den wir reservieren müssen um eine Anzeigeseite größer sein muss, als wir sonst tatsächliche Bilder puffern wollen. Aus diesem Grund haben wir jetzt einen Speicher mit 21 x 768.000 Byte sprich 15,3 MByte.

```
unsigned char frame[21][960000] __attribute__ ((section (".bss2")));
```

Diesen definieren wir global, da wir ihn über die gesamte Laufzeit des Programms benötigen. Mit __attribute__ ((section (".bss2"))) teilen wir weiters dem Linker mit, dass diese Variable im externen RAM platziert werden soll. Alle Variablen, die wir nicht kennzeichnen, werden im internen RAM abgelegt.
Weiters definieren wir uns noch ein zusätzliches Array mit ebenso vielen Elementen wie wir Anzeigeseiten puffern wollen.

unsigned short Inhalt[20];

Dieses Array initialisieren wir mit Nullen. Wir beschränken uns bei den Dateien, die wir anzeigen wollen, auf Dateien mit den Dateinamen auf der SD-Karte mit 1 – 65535.raw. So können wir uns durch ein einfaches Array kennzeichnen, welche Datei sich in einem Puffer befinden. 0 Bedeutet der Puffer ist leer.
Weiters hat diese Nomenklatur den Vorteil, dass wir bei den Links ebenfalls nur eine Nummer und keinen Namen speichern müssen.

5.9.3 *Displaycontroller*

Nun könnten wir eigentlich bereits den Displaycontroller konfigurieren. Doch zuvor müssen wir uns noch Gedanken darüber machen, wie die Prioritäten auf dem internen AHB1 Bus aussehen müssen.

5.9.3.1 *Prioritäten am AHB1*

Prinzipiell lassen sich hier zwei Busvergabesysteme einstellen. Zum einen besteht die Möglichkeit, die Busaufteilung über ein Round-Robin Verfahren zu vergeben, zum anderen über einen Prioritätenmechanismus. Wir mussten bei der Softwareentwicklung feststellen, dass das Display bei uns den Prioritätsmechanismus benötigt und dass dabei unser Display die höchste Priorität haben muss. Andernfalls, wenn z.B. die CPU auf das externe RAM zugreift und dabei größere Datenmengen transferieren muss, kommt es zu Aussetzern und Artefakten auf dem Display. Aus diesem Grund haben wir auf dem AHBCFG1 Register folgende Prioritäten vergeben:

Tabelle 9: Prioritäten auf AHB1

Priorität	Beschreibung	Wert auf AHBCFG1
1	LCD	5
2	CPU	4
3	GPDMA	3
4	AHB2	2
5	USB	1

Das bedeutet, dass unser Display die höchste Priorität besitzt, gefolgt vom Prozessorkern bis zum Schluss der USB Controller kommt, der die niedrigste Priorität besitzt. Aus dieser Tabelle ergibt sich für das AHBCFG1 Register ein Wert von 0x5123 4114.
Da nun die Vorarbeiten abgeschlossen sind, können wir damit beginnen, den LCD – Controller zu konfigurieren. Als erstes müssen wir – ähnlich wie beim externen RAM – zuerst die Prozessorpins, die für die Ansteuerung zuständig sind, konfigurieren.

5.9.3.2 *Pinkonfiguration*

Die nachfolgende Tabelle fasst alle für die Ansteuerung notwendigen Pins zusammen:

Tabelle 10: Display-Pin Zuordnung

Displ.	Pin	Displ.	Pin	Displ.	Pin
R5	P2.9	G5	P1.25	B5	P1.29
R4	P2.8	G4	P1.24	B4	P1.28
R3	P2.7	G3	P1.23	B3	P1.27
R2	P2.6	G2	P1.22	B2	P1.26
R1	P2.12	G1	P1.21	B1	P2.13
R0	GND	G0	P1.20	B0	GND
CLK	P2.2	DE	P2.4		

Nun müssen wir für jeden der verwendeten Pins den richtigen Wert für das PINSEL Register heraussuchen. Achtung: Für manche Pins muss 11b für andere 01b in das PINSEL Register geschrieben werden.
Nachfolgend die Tabelle für die PINSEL Register. Die Werte für das PINMODE Register ergeben sich, indem in alle für das Display verwendeten Pins der Wert 10b eingetragen wird (Pull up/down deaktiviert):

Tabelle 11: Display Port 2

Betreffender Pin	P2.15		P2.14		P2.13		P2.12		P2.11		P2.10		P2.9		P2.8	
Bit im PINSEL 4	Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	Bit 23	Bit 22	Bit 21	Bit 20	Bit 19	Bit 18	Bit 17	Bit 16
Wert					0	1	0	1					1	1	1	1
Betreffender Pin	P2.7		P2.6		P2.5		P2.4		P2.3		P2.2		P2.1		P2.0	
Bit im PINSEL 4	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Wert	1	1	1	1			1	1			1	1				

Tabelle 12: Display Port 1

Betreffender Pin	P1.31		P1.30		P1.29		P1.28		P1.27		P1.26		P1.25		P1.24	
Bit im PINSEL 3	Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	Bit 23	Bit 22	Bit 21	Bit 20	Bit 19	Bit 18	Bit 17	Bit 16
Wert					0	1	0	1	0	1	0	1	0	1	0	1
Betreffender Pin	P1.23		P1.22		P1.21		P1.20		P1.19		P1.18		P1.17		P1.16	
Bit im PINSEL 3	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Wert	0	1	0	1	0	1	0	1								

Die nicht eingetragenen Werte dürfen wir nicht berühren, das bedeutet, wir müssen die PINSEL und PINMODE Register einlesen, ausmaskieren, setzen und zurück schreiben. Wir wollen das hier am Beispiel von PINSEL4 veranschaulichen:
Zuerst löschen wir alle Bits im PINSEL4 Register, die Null sein müssen, das bedeutet wir verknüpfen das PINSEL4 Register mit einer „und“ Verknüpfung mit dem Wert 0xF5FF FFFF. Damit bleiben alle Bits, bis auf Bit 27 und Bit25 unverändert, nur diese beiden werden gelöscht. Anschließend verknüpfen wir das PINSEL4 Register mit einer „oder“ Verknüpfung mit dem Wert 0x050F F330. Damit werden genau die Bit, die bei uns 1 sind, gesetzt – alle anderen Bit im PINSEL4 Register bleiben unberührt. Im Code sieht dieses Segment wie folgt aus:

```
PINSEL4&=0xF5FF FFFF
PINSEL4|=0x50ff330;
```

Auf die selbe Art und Weise werden auch die Register PINSEL3, PINMODE3 und PINMODE4 gesetzt. Damit der LCD Controller Zugriff auf die Prozessorpins bekommt, muss noch ein weiteres zusätzliches Register konfiguriert werden. Das ist das PINSEL11 Register. Mit diesem wird konfiguriert, dass der LCD Controller auf

die Portpins zugreifen kann, und auf welche er Zugriff hat (Am Prozessor können verschiede Arten von Displays verwenden werden, die unterschiedliche Steuerleitungen benötigen). Daher müssen wir das Register so programmieren, dass wir ein TFT Display mit 16Bit verwenden. Zu diesem Zweck schreiben wir den Wert 0x0B auf das Register (LCD port enabled, TFT 16bit 5Red, 6Green, 5Blue).

5.9.3.3 *Timing & Enable*

Nun sind die Portleitungen fertig konfiguriert und wir können den Displaycontroller selbst konfigurieren.
Zuerst müssen wir dazu den LCD Controller mit Spannung versorgen. Dies geschieht, indem wir im PCONP Register das PCLCD Bit setzen. Dadurch wird der Displaycontroller mit Betriebsspannung versorgt. Anschließend konfigurieren wir den Pixelclock für das Display. Dazu konfigurieren wir unser System auf den höchst möglichen Pixelclock, den es generieren kann (24MHz) und schreiben dazu den Wert 0x02 auf das LCD_CFG Register. Weiters müssen wir noch BCD Bit im LCD_POL Register setzen, sodass der optionale Clockdivider disabled ist. Weiters setzen wir in diesem Register die Anzahl der Spalten fest und dass der Pixelclock invertiert gehört.
Danach setzen wir die Register LCD_TIMH und LCD_TIMV für die horizontalen und vertikalen Timing Werte aus dem Display Datenblatt.
Zuletzt müssen wir noch festlegen, wo sich der Speicherbereich befindet, der die anzuzeigenden Daten enthält und den Puffer für das Display darstellt. Das ist der zuvor erzeugte Speicher im externen RAM.

```
LCD_UPBASE = frame[20];
```

Schließlich können wir das Display einschalten, indem wir LCD_CTRL |= LCD_CTRL_LcdEn setzen. Ab sofort wird der von uns zugeteilte Speicherbereich im externen RAM am Display dargestellt.

5.9.4 *Anzeige von Daten*

Der prinzipielle Aufbau unseres Bildes sieht folgendermaßen aus. Ein Anzeigepuffer ist 768.000 Byte groß. Jeweils 1.600 Byte entsprechen einer Zeile und jeweils 2 Byte entsprechen einem Pixel. Die Farbaufteilung in jeweils zwei Byte sieht wie folgt aus:

Tabelle 13: Display Datenformat

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Farbe	R5	R4	R3	R2	R1	G5	G4	G3	G2	G1	G0	B5	B4	B3	B2	B1

Soll ein Pixel weiß auf dem Display dargestellt werden, so muss man 0xFFFF auf den entsprechenden Speicherplatz schreiben, soll es schwarz sein 0x0000.
Nun haben wir unser Programm folgendermaßen aufgebaut. Der Anzeigepuffer für den Displaycontroller bleibt immer derselbe. Soll eine neue Seite von der SD-Karte geladen werden, so wird der Inhalt immer zuerst auf den Anzeigepuffer geladen (damit man den Bildaufbau und damit den Fortschritt sieht) und anschließend auf eine zweite Speicherstelle kopiert. Somit kann jederzeit direkt im Anzeigepuffer etwas auf dem Hintergrund dargestellt, aber auch wieder gelöscht werden (Ist eine Seite bereits in einem Puffer, muss diese natürlich nicht neu von der Speicherkarte gelesen werden, sondern kann direkt auf den Anzeigepuffer zurückkopiert werden).

Um nun Elemente am Display anzeigen zu können, haben wir ein paar Funktionen gebaut:

5.9.5 *Funktionen*

5.9.5.1 *lcd_point*

void lcd_point(int x, int y, unsigned c)
Diese Funktion zeichnet einen Punkt am Display. Auf diese Funktion bauen die nachfolgenden Funktionen auf. Weiters prüft diese Funktion, ob sich der Punkt auf dem Display befindet. Damit ist sichergestellt, dass kein falscher Speicherbereich überschrieben wird.

```
if (x < H_SIZE && y < V_SIZE)
   {
     unsigned short *zeiger = ((unsigned short *)frame[20]) + (H_SIZE * y) + x;
     *zeiger = c;

   }
```

5.9.5.2 *delete_lcd_point*

void delete_lcd_point(int x, int y)

Diese Funktion löscht genau einen Punkt am Anzeigepuffer. Dazu kopiert diese Funktion den Punkt von der zweiten Speicherstelle auf den Ausgabepuffer

```
if (x < H_SIZE && y < V_SIZE)
  {
        unsigned short *zeiger_1 = ((unsigned short *)frame[20]) + (H_SIZE * y) + x;
        unsigned short *zeiger_2 = ((unsigned short *)frame[curr_frame]) + (H_SIZE * y) + x;

    *zeiger_1 = *zeiger_2;

  }
```

5.9.5.3 *drawrectline*

void drawrectline(int x, int y, int w, int h, unsigned c)

Diese Funktion zeichnet ein leeres Rechteck, welches nicht ausgefüllt ist.

5.9.5.4 *delete_drawrectline*

void delete_drawrectline(int x, int y, int w, int h, unsigned c)

Diese Funktion löscht genau die Linien, die von drawrectline gezeichnet wurden

5.9.5.5 *drawbar*

void drawbar(int x, int y, int w, int h, int percent, unsigned c)

Diese Funktion zeichnet einen Prozentbalken.

5.9.5.6 *delete_bar*

void delete_bar (int x, int y, int w, int h, int percent, unsigned c, unsigned bc)

Diese Funktion löscht genau den Bereich, den drawbar() gezeichnet hat.

5.9.5.7 *drawchar*

int drawchar(int x, int y, char ch, unsigned c, unsigned bc)

Diese Funktion zeichnet einen Buchstaben auf das Display. Dazu haben wir von jedem Buchstaben eine Pixelgrafik erstellt, und in einer Matrix gespeichert, ob für den Buchstaben zur Anzeige das jeweilige Pixel gesetzt gehört oder nicht. Weiters haben wir noch die Buchstabenbreite mitgespeichert. Diese Schriftdaten werden beim Programmstart von der SD-Karte geladen und in das externe RAM kopiert. Soll nun ein Buchstabe angezeigt werden, so wird der passende Datensatz für den Buchstaben geladen und für alle Bit, die in diesem Datensatz gesetzt sind wird am Display ein Punkt in der Farbe c gezeichnet, für alle nicht gesetzten Bit ein Punkt der Farbe bc.
Eine spezielle Funktion besitzt die Farbe 0xfe7f. Diese wird als „durchsichtig“ dargestellt. Das bedeutet, dass für Punkte dieser Farbe einfach keine Ausgabe auf dem Display stattfindet. Weiters gibt die drawchar() Funktion die Breite des Buchstaben zurück. Dies ist von Nutzen, wenn man wieder einen Text vom Dipslay entfernen möchte, damit man weiß, wie groß der Bereich ist, den man wiederherstellen muss.

5.9.5.8 *lcd_putString*

int lcd_putString(int x, int y, char *pStr, unsigned c, unsigned bc)
Diese Funktion schreibt einen ’\0’ terminierten String mit Hilfe der Funktion drawchar() auf das Display. Weiters addiert die Funktion die Returnwerte der drawchar() Funktion, die die Breite des gezeichneten Buchstaben auf dem Display darstellen. Anschließend gibt die Funktion diese Summe als eigenen Returnwert zurück.

Wir sind der Meinung, dass sich durch diese Funktionen alle wichtigen Darstellungsobjekte auf dem Display zeichnen lassen. Sollte trotzdem der Fall auftreten, dass eine weitere Zeichenfunktion benötigt wird, so lässt sich diese einfach – durch Modifikation einer bereits implementierten Funktion – erstellen. Das Ausgabeprinzip auf den tatsächlichen Ausgabepuffer über den Pointer und der Berechnung von X und Y Koordinate ist in allen Fällen dasselbe. Weiters kann auch immer auf die Funktionen lcd_point und delete_lcd_point zurückgegriffen werden. Diese Funktionen prüfen bei der Ausgabe auf das Display, ob die angegebenen

Koordinaten gültig sein können, und verwerfen die Ausgabe im Fehlerfall, sodass kein unbeabsichtigter Speicherzugriff erfolgen kann.

5.9.6 *Speicherverwaltung*

Wie bereits zuvor erwähnt, haben wir den reservierten Speicher im RAM so groß gewählt, dass wir 20 Bilder puffern können. Das bedeutet, dass sobald wir ein Bild zum ersten Mal von der SD-Karte geladen haben, dieses im Arbeitsspeicher bleibt, auch wenn wir ein neues Bild wählen. Dazu benötigen wir aber eine Speicherverwaltung. Diese Speicherverwaltung sieht folgendermaßen aus:
Um zu wissen welche Seiten wir überhaupt im RAM haben, haben wir zuvor das Array `unsigned short Inhalt[20]` definiert. Sobald wir eine Seite ins RAM laden, tragen wir den Dateinamen in dieses Array ein. Da unser RAM aber irgendwann auch voll sein kann und wir danach eine Seite auslagern müssen, führen wir einen zusätzlichen Speicherplatz für jede Seite ein, auf dem wir festhalten, wie lange schon nicht mehr auf die Seite zugegriffen wurde. Wir erweitern also unser Inhalt[] Array wie folgt:

```
unsigned short Inhalt[20][2]
```

Das bedeutet, dass wir für jeden unserer 20 Anzeigepuffer zwei Werte mitspeichern. Am Speicherplatz Inhalt[x][0] den Seitennamen, am Speicherplatz Inhalt[x][1] einen Zähler. Diesen Zähler setzen wir, sobald die jeweilige Seite aufgerufen wird (entweder von der SD-Karte geladen wird, oder nur auf den Anzeigepuffer kopiert wird) auf 65 535. Jedes Mal wenn sich die angezeigte Seite ändert, werden diese Counter aller 20 gepufferten Seiten dekrementiert (Außer wenn diese 0 sind, dann bleiben diese unverändert).
Soll nun eine neue Seite geladen werden sieht der Ablauf wir folgt aus:

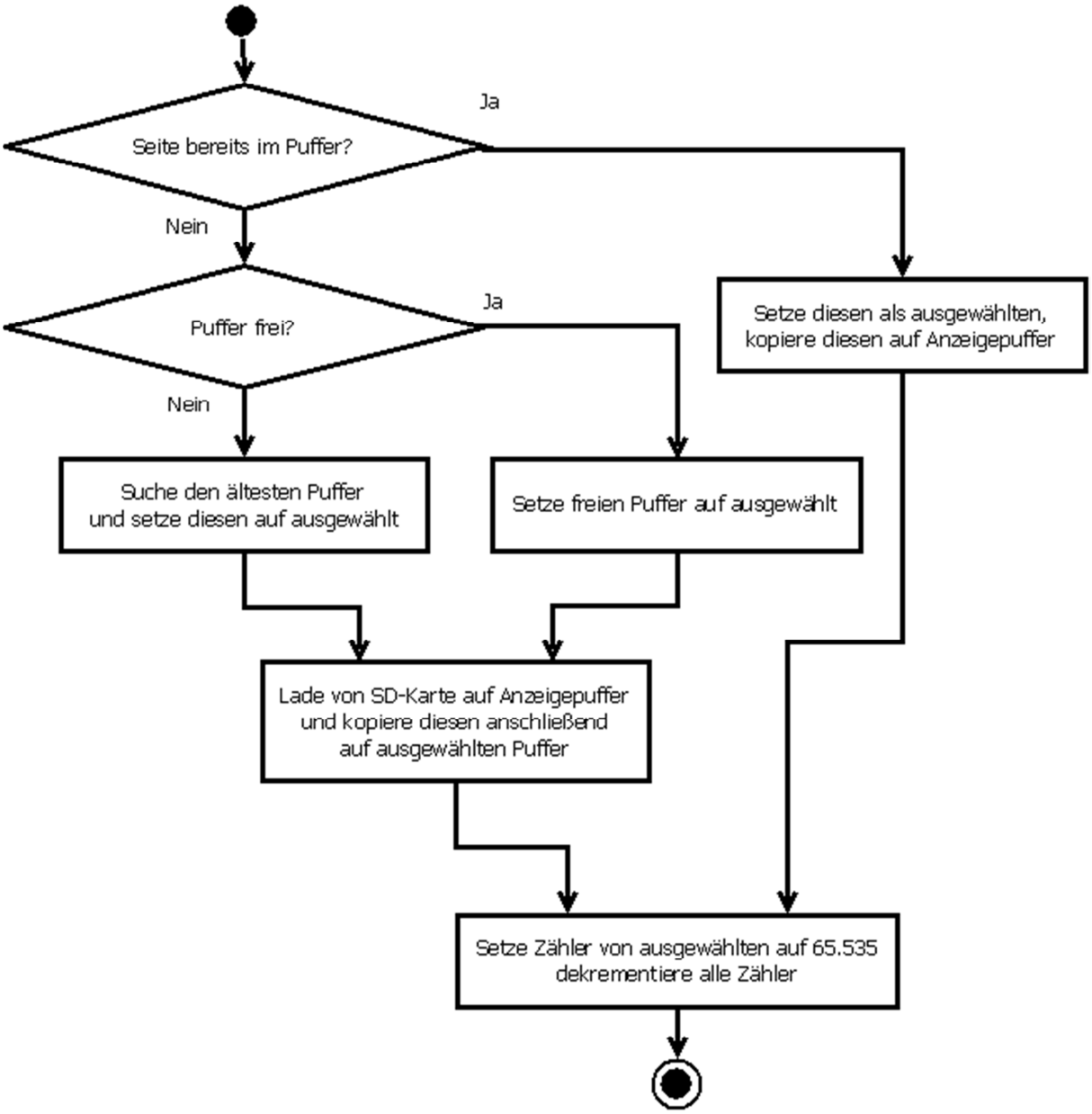

Abbildung 13: Flussdiagramm Speicherverwaltung

Zuerst wird von allen 20 Seiten der Dateiname überprüft, um festzustellen, ob dieser bereits in einem der Pufferspeicher ist. Ist dem der Fall, so wird diese Seite in den Anzeigepuffer kopiert, es wird gespeichert, welcher Puffer gerade angezeigt wird, der Counter dieser Seite wird auf 65.535 gesetzt und alle Counter werden dekrementiert. Ist die gesuchte Seite noch nicht gepuffert, so wird anhand der Dateinamen der gespeicherten Seiten überprüft, ob es noch einen Speicher gibt, der gar nicht belegt ist. Tritt dieser Fall ein, so wird dieser freie Puffer als der angezeigte gespeichert, die Grafik wird von der SD-Karte zuerst auf den Ausgabepuffer kopiert, anschließend der Ausgabepuffer auf den gespeicherten Puffer dupliziert. Daraufhin wird wieder der

Zähler des ausgewählten Puffers auf 65.535 gesetzt und die Zähler aller Puffer decrementiert. Weiters wird natürlich auch der Dateiname des ausgewählten Puffers gesetzt.
Im dritten und letzten Fall ist die gesuchte Seite noch nicht im Speicher und es ist kein Pufferspeicher mehr leer. Dann wird der Pufferspeicher gewählt und als ausgewählter Speicher markiert, dessen Zähler den niedrigsten Wert hat, bzw. der erste Puffer der den Wert 0 hat. Damit wird also immer die Seite ausgelagert, die schon am längsten nicht mehr gebraucht wurde. Anschließend wird wieder – wie im Fall 2 – die Grafik von der SD-Karte auf den Ausgabepuffer kopiert. Dieser Ausgabepuffer wird anschließend wieder auf den ausgewählten dupliziert, der Zähler des ausgewählten Puffers rückgesetzt und alle Zähler dekrementiert. Natürlich muss auch hier der Dateiname des ausgewählten Puffers aktualisiert werden.

5.10 *Touchscreen*

5.10.1 *Allgemein*

Um Eingaben in unser Gerät tätigen zu können, haben wir als Mensch-Maschinen Interface einen Touchscreen gewählt. Dessen Auswertung möchten wir nun noch etwas erörtern. Wie wir bereits im Hardwarekapitel erklärt haben, handelt es sich bei unserem Touchscreen um einen resistiven Touchsreen, den wir mit Hilfe des prozessorinternen ADC auswerten wollen. Dazu müssen wir als erstes den prozessorinternen ADC konfigurieren.

5.10.2 *ADC Konfiguration*

Hierbei handelt es sich um einen analog nach digital Konverter (ADC), der nach dem Prinzip der schrittweisen Annäherung arbeitet (sukzessive Approximation) Für eine Konvertierung benötigt dieser 11 interne Taktzyklen und 2.44µs. Daher müssen wir den Frequenzteiler für den ADC so einstellen, dass dieser eine Frequenz von 4.5MHz erzeugt. Zuvor müssen wir allerdings auch den ADC mit Betriebsspannung versorgen. Dazu müssen wir im PCONP das PCAD Bit setzen.
Da unser Peripherieclock gleich dem Prozessorclock / 4 ist, also 18MHz, müssen wir für den ADC noch einmal einen Teiler von 4 einstellen. Dies geschieht im AD0CR

Register. Weiters müssen wir hier das PDN Bit setzen, sodass der ADC aktiviert wird.
Anschließend können wir beginnen, den Touchscreen auszuwerten. Das Verfahren selbst wurde ja schon im Hardwareteil dieser Arbeit erörtert. Trotzdem möchten wir hier noch einmal auf die softwaremäßige Auswertung zurückkommen.

5.10.3 *Messung*

Eine Touchscreenauswertung besteht immer aus vier Messungen.
1.Messung
Die Spannung wird von links nach rechts (also am X Kanal) auf die touchsensitive Folie angelegt. Dazu werden die beiden dafür notwendigen Portpins als Output definiert, der linke auf Plus geschaltet, der rechte auf Masse. Anschließend wird das Ergebnis am Y-Messkanal mit Hilfe des ADC erfasst und auf der Variable X1 gespeichert. Danach wird auch der auf Plus geschaltete Pin wieder auf Masse geschaltet, da, wenn dieser direkt auf Messeingang für den ADC umgeschaltet wird, es sehr lange dauert, bis sich die Spannung auf dem Pin abbaut und die nächste Messung sonst verfälscht wäre.
2.Messung
Die beiden für den X Kanal zuständigen Pins werden als Eingang und die für den Y Kanal zuständigen Pins als Ausgang definiert. Weiters wird der obere Pin auf Versorgungsspannung, der untere auf Masse geschaltet. Nun kann am X-Messkanal ein neuer Wert gemessen werden. Dieser wird auf der Variable Y1 gespeichert. Nun muss der Pin, der auf Versorgungsspannung geschaltet war, wieder auf Masse geschaltet werden, um die Ladung abzubauen.
3.Messung
Diese läuft genau wie Messung eins ab, mit dem einzigen Unterschied, dass Versorgungsspannung und Masse am X-Kanal vertauscht werden. Anders ausgedrückt, die Spannung liegt jetzt von rechts nach links an der Touchfolie an.
Dieses Ergebnis speichern wir auf der Variable X2
4.Messung
Diese funktioniert erneut so wie Messung 2, nur dass dieses Mal die Polaritäten am Y-Kanal vertauscht sind.
Dieses Ergebnis speichern wir auf der Variable Y2.

5.10.4 *Berechnung*

Nun müssen wir als Erstes detektieren, ob der Touch überhaupt gedrückt wurde. Dazu müssen wir einfach die Variablen X1 und X2 zusammenzählen, sowie die Variablen Y1 und Y2. Die Summe dieser Werte muss nahe dem Maximalwert des ADCs liegen, also nahe dem Wert 1024.
Nun können wir, indem wir an diesem Vergleichswert etwas variieren, die Empfindlichkeit des Touchscreens einstellen. Machen wir den Wert kleiner, so reagiert der Touch schon bei weniger Druck – die Auswertung wird aber ungenauer. Erhöhen wir diesen Wert, muss man sehr fest auf die Touchfolie drücken um eine Auswahl zu tätigen. Bei unseren praktischen Versuchen hat sich für diese Konstante der Wert 910 als sehr gut geeignet herausgestellt. Weiters lässt sich durch diesen Wert auch eine gewisse Aussage über den Druck, der auf das Display ausgeübt wurde, machen. Jedoch ist diese Auswertung sehr ungenau, da hierbei noch sehr viele andere Faktoren beeinflussend sind (Querschnitt, mit dem auf das Display gedrückt wird, Position, …).
Die aktuelle Position könnten wir nun folgendermaßen berechnen:
Wir zeigen hier die Berechnung an den X-Koordinaten, die Y-Koordinaten berechnen sich analog:
X Position = (X1 + (1023 – X2)) /2
Zur Erklärung: X1 ist die eigentliche Abweichung von links nach rechts. X2 ist die Abweichung von rechts nach links. Wenn wir also Maximalwert – X2 rechnen, muss dieser rechnerisch wieder X1 ergeben. Wenn wir diese beiden Werte also addieren, müssen wir die Summe wieder durch zwei dividieren, um die korrekte Abweichung zu bekommen.
Wenn wir dieselbe Berechnung auch noch für die Y Punkte durchführen, so haben wir die Koordinate, an dem der Touch betätigt wurde, berechnet.

6 Ergebnisse der Arbeit

6.1 *Prototyp*

Bisher haben wir uns in unserer Arbeit hauptsächlich auf Hardwaregrundlagen und Softwaretreiber spezialisiert. Um die Fähigkeiten unserer Plattform zu demonstrieren, haben wir einen voll funktionsfähigen Prototypen konstruiert. Dieser soll dazu dienen, die Möglichkeiten unserer Demonstrations-plattform aufzuzeigen, und den Anreiz zu schaffen, diese für praktische Projekte einzusetzen. Folglich soll es damit auch möglich sein, dieses Gerät als Plattform für die Softwareentwicklung spezieller Interfaces zu verwenden.

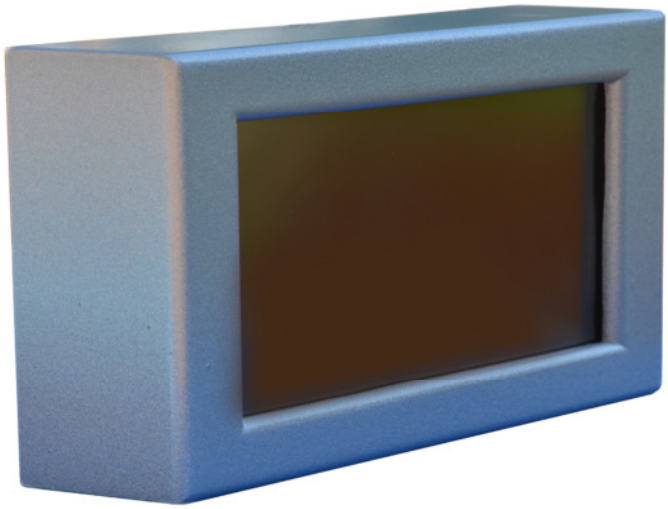

Abbildung 14: Der Prototyp

Wie wir im Hardwareabschnitt der Arbeit erläutert haben, besteht der Kern unserer Arbeit aus der von uns entworfenen Leiterplatte, welche als Herzstück einen ARM7 Prozessor von NXP beinhaltet. Weiters verfügt unser Gerät über einen SD-Karten Slot, über den zum einen die für den Betrieb notwendige Firmware eingespielt und so das prozessorinterne Flash programmiert werden kann. Zum anderen ist diese Schnittstelle aber auch für den tatsächlichen Betrieb essentiell. So werden von hier die im Betrieb notwendigen Anzeigen und Konfigurationsdaten geladen. Weiters verfügt unser Prototyp über eine CAN-Schnittstelle, die voll funktionsfähig ist. Diese

ist dazu gedacht, den Prototypen tatsächlich zur Entwicklung eines Bedieninterfaces für andere Anlagen nutzen zu können.
Zur Bedienung haben wir auf unserem Prototypen das 7“ TFT Touchdisplay, welches wir im Hardwareabschnitt beschrieben haben, verbaut. Dieses Display – mit den damit verbundenen Features – soll das Hauptaugenmerk unseres Prototypen sein. Um die Fähigkeiten unseres Prototypen zu demonstrieren, haben wir eine Musterapplikation geschrieben. Diese soll als Beispiel für eine mögliche Bedienoberfläche dienen, wenn unser Entwurf für einen tatsächlichen Einsatz adaptiert wird.

6.2 *Probleme bei herkömmlichen Ansätzen*

In dieser Musterapplikation haben wir einen alternativen Ansatz für das Mensch-Maschinen-Interface gewählt. In herkömmlichen Applikationen ist es meist so, dass dann wenn ein Touchscreen eingesetzt wird, dieser sowohl zum Markieren von Objekten, als auch zum Auswählen von Objekten eingesetzt wird. Damit gibt es einerseits die Möglichkeit, dass schon die erste Berührung als Auswahl gewertet wird. Dadurch passiert es aber oft, dass eine falsche Auswahl getroffen wird, da vor allem bei nur leichtem Druck die Genauigkeit der Auswertung von resistiven Touchscreens leidet. Die zweite Möglichkeit, wie die Auswahlproblematik gelöst sein kann ist, dass diese durch einen Doppelklick bestätigt werden muss. Doch auch hier entsteht genau dasselbe Problem. Da bei einem Doppelklick zwei Berührungen rasch hintereinander durchgeführt werden müssen, kann es zum einen passieren, dass diese entweder von der Position her ungenau sind, oder nur durch sehr leichte Berührungen ausgeführt werden. Alles in allem kommt es dadurch auch bei diesem Auswahlmechanismus häufig zu Erkennungsfehlern.
Ein weiteres Problem, das eigentlich bei beiden Eingabemethoden besteht, ist, dass keine Vorauswahl dargestellt werden kann. Vor allem für Menschen mit Sehschwächen oder auch motorischen Schwächen wäre es von Vorteil, wenn sie – schon bevor sie die Auswahl treffen – diese Auswahl visuell bestätigt bekommen könnten. Das ist bei einer herkömmlichen Computermaus selbstverständlich. Hier sieht man außerdem, wo sich der Mauszeiger am Bildschirm befindet, und hat somit optisch seine Auswahl schon hervorgehoben. Mit einem „klick“ bestätigt man diese anschließend nur noch, und hat hierbei sogar noch das zusätzliche haptische Feedback. Doch auch bei diesem Verfahren gibt es einen Nachteil. Diesem fehlt –

abgesehen davon, dass man schon mal ein zusätzliches Eingabegerät benötigt – eine wichtige Eigenschaft. Bei diesem Verfahren finden die Visualisierung und die Auswahl an zwei unabhängigen Orten statt. Die Positionierung mit der Maus basiert auf einem relativen Bezugssystem zum Ausgangspunkt. Bei einer Touchscreen-Bedienoberfläche ist das anders. Hier haben wir eine absolute Positionierung auf der Touchfolie. Weiters deckt sich diese Touchfolie mit der Anzeige. Dadurch ist die Bedienung über die Tocheingabe wesentlich intuitiver. Weiters ist es in vielen Anwendungsfällen auch gar nicht möglich, eine Computermaus als Eingabegerät zu installieren. Ist das Gerät ein Bedieninterface für eine Anlage, die irgendwo auf einer Wand montiert ist, so kann hierbei nicht einfach eine Computermaus daran befestigt werden.

6.3 *Das Eingabekonzept*

Um nun die positiven Eigenschaften von Maus und Touchscreen zu vereinen, haben wir ein neues Konzept zur Benutzereingabe entworfen und umgesetzt.
Bei diesem Konzept findet sowohl die Positionierung als auch die Auswahl über den Touchscreen statt. Damit haben diese beiden Schritte in der Bedienung einen absoluten Positionszusammenhang und sind somit sehr intuitiv. Zusätzlich sind diese Schritte jedoch nicht in einem Interaktionsschritt zusammengefasst, sondern gliedern sich in zwei Aktionen auf. Zum einen die Zielwahl und zum anderen die eigentliche Auswahl. Weiters versuchten wir, diese beiden Schritte soweit zu entkoppeln, dass, sobald etwas markiert ist und anschließend die Auswahl stattfindet, sich diese Auswahl nicht mehr verändern darf. Damit soll sichergestellt werden, dass dann, wenn eine Markierung getroffen wurde, diese auch tatsächlich ausgewählt wird. Somit soll die Quote von Falsch-Auswahlen gesenkt werden.
Um diese Zielsetzung zu erreichen, werten wir die Eingabe über unser Display über zwei Wege aus. Zusätzlich zwingen wir dadurch den Benutzer/-in einen „kräftigen“ und damit deutlichen Druck auf das Display auszuüben.

6.4 *Konstruktion*

Der erste Eingabeweg ist der, den man erwarten würde. Hier werten wir die Touchfolie selbst mit unserem Prozessor und dem darin integrierten ADC aus. Damit können wir den Punkt, an dem das Display bedient wird, errechnen. Zum Zweiten haben wir bei der mechanischen Konstruktion unseres Prototypen vorgesorgt, indem

wir nicht wie üblich unser Display starr verschraubt – sondern dieses beweglich montiert haben. Zu diesem Zweck haben wir dieses, auf den sonst für die Montage vorgesehenen Punkten, auf Microtaster montiert.

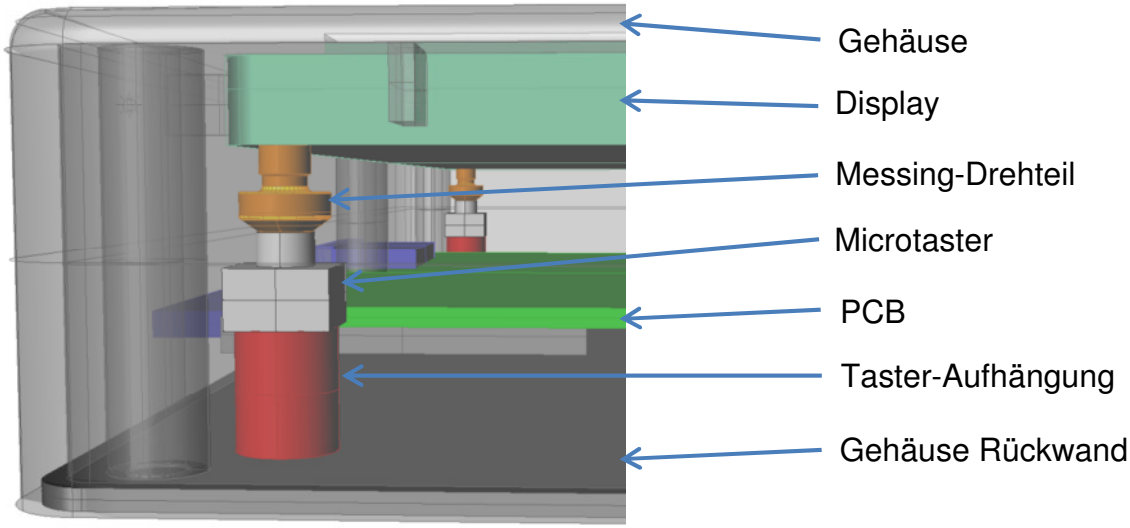

Abbildung 15: Display Aufhängung

Somit kann sich nun das Display in einer gewissen Führung bewegen. Sobald genügend Druck auf die Touchfolie ausgeübt wird, wird der Druckpunkt der Taster erreicht und das Display gibt unter einer deutlich spürbaren Bewegung nach hinten nach. Diese Bewegung ist natürlich reversibel sobald der Druck auf das Display wieder nachlässt. Der Vorteil durch diese Konstruktion über Taster ist, dass dadurch eine Hysterese im ausgeübten Druck entsteht, um den Mechanismus auszulösen und bis dieser wieder in die Ausgangsposition zurückgeht.

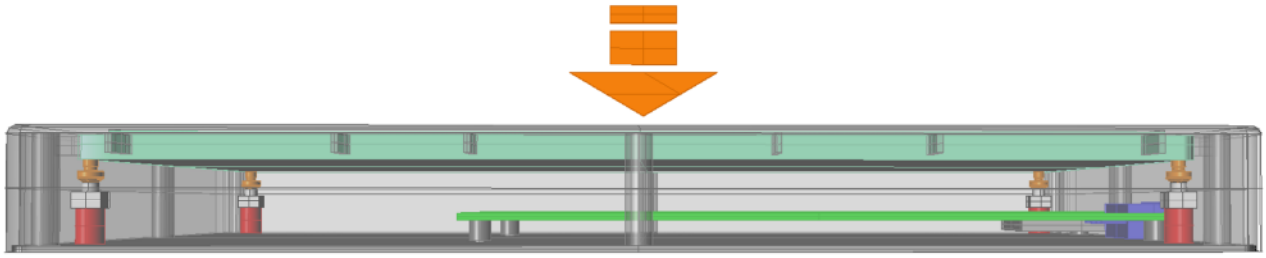

Abbildung 16: Display Montage

6.5 *Auswertung*

Da wir nun zwei getrennte Quellen zum Erfassen der Eingabe haben, besitzen wir die Möglichkeit, dies auch getrennt für das Markieren und die Auswahl zu benutzen. Aus diesem Grund haben wir nun das Signal des Touchscreens benutzt, um Elemente zu markieren, und das Signal der Taster, um eine Auswahl zu treffen. Softwaretechnisch haben wir das in unserer Musterapplikation folgendermaßen eingebaut.

Beim Programmstart wird die Datei 1.raw von der Speicherkarte eingelesen und der Inhalt ins externe RAM kopiert. Dabei wird genau die Speicherverwaltung verwendet, die wir im Kapitel Software erklärt haben. Diese, von der SD-Karte in die erste Speicherstelle kopierten Daten, werden dabei vom Displaycontroller dargestellt. Zusätzlich sind mit dieser Anzeigedatei auf der SD-Karte noch sensitive Bereiche definiert. Diese sensitiven Bereiche sind Links auf andere Seiten. Insgesamt sind in unserer Demonstrationssoftware bis zu 16 Links pro Seite definierbar. Diese Begrenzung ist beliebig gewählt, und kann bei Bedarf einfach erweitert werden. Jeder solcher Link besteht aus zwei X-Koordinaten, zwei Y-Koordinaten und einer zwei Byte großen Zahl, welche den Dateinamen des Ziels darstellt. Damit lässt sich also ein rechteckiger Bereich definieren, welcher einen Link auf eine beliebige Datei darstellt.

6.6 *Musterapplikation*

Nun zu unserem eigentlichen Programmablauf. In unserem Programm wird ständig der Touchscreen mit dem ADC vermessen. Sobald der Touchscreen berührt wird, berechnen wir die Position, an der die Eingabe stattfindet. Anschließend überprüfen wir, ob diese Position in der im Augenblick angezeigten Grafik als sensitiver Punkt definiert ist. Ist dies der Fall, so zeichnen wir mit unseren zuvor besprochenen Displayfunktionen ein Rechteck über diesen sensitiven Bereich.

Abbildung 17: Musterapplikation ohne Auswahl

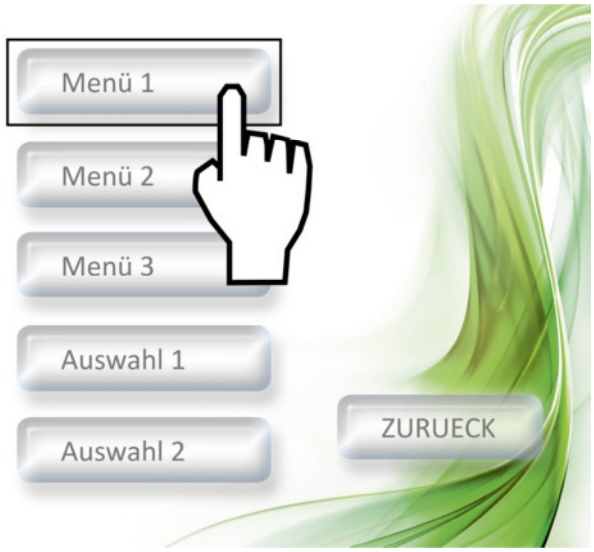

Abbildung 18: Musterapplikation mit Auswahl und visuellem Feedback

Findet als Nächstes eine Eingabe in einem Bereich statt, der außerhalb dieses sensitiven Bereiches liegt, so löschen wir diesen wieder. Wenn direkt die Eingabe von einem sensitiven Bereich in den nächsten wechselt, so löschen wir die alte Markierung und setzen zugleich die neue. Dadurch ist es sehr einfach, gezielt einen Bereich auszuwählen, da das System ständig über seine Markierung informiert. Somit ist die Gefahr einer Falschauswahl schon das erste Mal reduziert. Nun zu der eigentlichen Auswahl. Hierfür verwenden wir die von uns gewählten Taster, die zur Montage des Displays dienten. Sobald einer dieser Taster gedrückt ist, werten wir

dies als Eingabe. Da man aber, um den Taster überhaupt betätigen zu können, das Display berühren muss, haben wir zugleich eine Markierung. Das bedeutet, dass wir immer zuerst eine Markierung haben, bevor der Taster gedrückt werden kann. Dies beruht darauf, dass die Taster einen größeren Druck benötigen als die Touchfolie. Weiters ist durch diesen hohen Druck auch gewährleistet, dass wir eine ordentliche Position haben. Bei nur sehr leichten Berührungen der Touchfolie kann es manchmal zu einer ungenauen Bestimmung der Position kommen. Diese Gefahr ist somit auch reduziert. Fassen wir also das Eingabekonzept zusammen: Sobald man das Display berührt, bekommt man, sofern man sich auf einem sensitiven Bereich befindet, diesen visuell dargestellt. Damit kann man seine Auswahl genau platzieren und hat dabei ein visuelles Feedback. Hat man nun den richtigen Bereich getroffen, erhöht man den Druck auf das Display. Damit steigt die Kraft, die auf die Taster wirkt, über deren Druckpunkt, und man bekommt die Auswahl durch ein haptisches Feedback bestätigt.
Sobald eine Auswahl auf einer Markierung getroffen wurde, zeigt das System die Zielseite entweder direkt aus dem RAM an (falls diese bereits gepuffert ist), oder lädt diese von der Speicherkarte. Weiters ändert das System dabei natürlich die sensitiven Bereiche mit den Links mit, sodass die neuen Verknüpfungen der neuen Seite wirksam werden. Die Konfiguration der Seiten und der Links erfolgt dabei rein über die SD-Karte und benötigt keine Eingriffe in die Software. Die Anzeigedaten lassen sich somit auch ohne jegliche Programmierkenntnisse erstellen.

Unsere fertige Plattform unterzogen wir natürlich mehreren Tests. So interessierten uns zum einen die Mächtigkeit unserer Applikation und zum anderen die Bedienbarkeit. Für diese Tests erstellten wir also Musterbilder, die eine Menüstruktur darstellten, und definierten darauf sensitive Flächen. Diese Testdaten spielten wir auf eine SD-Karte und testeten damit unsere Plattform.

7 Abschließende Tests

Die präsentierten Erkenntnisse basieren auf Tests, die wir im Zuge der Entwicklung mit Personen aus dem Kollegen-, Freundes- und Familienkreis durchgeführt haben. Ziel dieser Arbeit war die Schaffung einer Entwicklungsplattform für spätere Projekte. Das Eingabekonzept, das wir entwickelt haben, konnten wir so ausreichend auf Intuitivität testen. Die Entwicklung einer praktischen Applikation mit Validierung der Einsatzfähigkeit bei den Zielgruppen, bedarf auf Grund des Umfanges, eines eigenständigen Projektes und war auch nicht Aufgabenstellung dieser Arbeit. Folgende Erkenntnisse konnten wir dennoch aus unseren Tests erlangen:

7.1 *Konfiguration*

Die Anzahl der Seiten ist praktisch beliebig erweiterbar. Dadurch, dass Speicherkarten mit Kapazitäten von mehreren Gigabyte zu sehr geringen Preisen zur Verfügung stehen, besteht keine praktische Begrenzung, was die Anzahl der möglichen Seiten betrifft. Auch mit der von uns willkürlich gewählten Begrenzung von 16 Links pro Seite kann locker das Auslangen gefunden werden. Wird diese Möglichkeit voll ausgeschöpft, so werden die einzelnen sensitiven Flächen schon sehr klein, was die physikalische Größe auf dem Display betrifft. Auch hat sich die Verwendung der von uns implementierten Pufferung als nützlich erwiesen. Dabei kann z.B. bei einem „Zurück“ Button eine sehr kurze, kaum merkbare Umschaltzeit am Display erreicht werden.
Weiters haben wir mit unserer Demonstrationsplattform auch genau das erreicht, was wir erreichen wollten, nämlich, dass der Erstellter von Menüstrukturen absolut keine Programmierkenntnisse benötigt. Die einzige Fähigkeit, die dafür benötigt wird, sind Grundlagenkenntnisse in der Grafikbearbeitung zum Erstellen der Bilder.

7.2 *Bedienbarkeit*

Auch was die Bedienbarkeit betrifft, konnten wir durch unser neu entworfenes Eingabekonzept sehr gute Ergebnisse erzielen. Es stellte sich heraus, dass es gerade durch die Entkopplung der Markierung und Auswahl, zu keinen Fehleingaben

mehr kam. Zum einen hilft hier mit, dass man zur Eingabe durch die Taster etwas fester auf das Display drücken muss um es auszulösen. Dadurch ist eine bessere Positionsbestimmung durch die Touchfolie gegeben, und es kann außerdem zu keinen irrtümlichen Eingaben kommen. Zum anderen hat man, bevor man die Auswahl noch trifft, bereits das optische Feedback über die Auswahl und kann diese gegebenenfalls korrigieren.

8 Ausblick

Man könnte in der Hardware den 3.3V Fixspannungsregler durch ein Schaltnetzteil ersetzen. Dadurch könnte Verlustleistung eingespart werden.

In der Softwareupdatefunktion könnte noch eine zusätzliche Überprüfung eingebaut werden, ob die Checksumme für die Einsprungvektoren OK ist. Im Augenblick könnte es passieren, wenn eine Software mit fehlerhafter Prüfsumme eingespielt wird, dass die Updatefunktion nicht mehr startet.

Eine Musterapplikation sollte programmiert werden, welche tatsächlich über den CAN Bus eine Bedienschnittstelle darstellt.

Der Prozessor LPC2478 könnte durch den LPC1788 ersetzt werden. Dabei handelt es sich um einen Nachfolger, der pinkompatibel ist. Der Nachfolger kann höher getaktet werden und besitzt eine neue interne Architektur (Cortex M3). Weiters verfügt dieses Modell dann auch über ein integriertes EEPROM Im Laufe unseres Projektes war dieser aber leider noch nicht verfügbar.

9 Abbildungsverzeichnis

10 Abkürzungsverzeichnis

ADC	Analog Digital Converter
BGA	Ball Grid Array - SMD Bauform bei dem die Pins unter den Chip angeordnet sind
CAN	Controller Area Network, Feldbusssystem
CCFL	Cold Cathode Fluorescent Lamp
CPSR	Current Program Status Register
DMA	Direct Memory Access
EEPROM	Electrically Erasable Programmable Read Only Memory
EMC	External Eemory Controller
FET	Feld Effekt Transistor
FFC	Felxible Flat Cabel
FIQ	Fast Interrupt Request
Flash	nicht flüchtiger Halbleiterspeicher
GUI	Graphical User Interface
HMI	Human Machine Interface
IDE	integrated development environment
JTAG	Joint Test Action Group
LED	Light Emitting Diode
LQFP	SMD Bauform bei dem die Pins run um den Chip angeordnet sind
LVDS	Low Voltage Differential Signaling
MAM	Memory Accelerator Module
MMS	Mensch Maschinen Schnittstelle
NOP	no operation
OpenOCD	Open On Chip Debugger
PC	Programm Counter
USB	Universal Serial Bus

11 Literatur

Laehyun Kim, Hyunchul Cho, Sehyung Park and Manchul Han – A Tangible User Interface with Multimodal Feedback – Lecture Notes in Computer Science 2007 – Pages 94-103 – LNCS 4552 – Juli 2007

Seung-Chan Kim , Tae-Hon Yang, Byung-Kil Han and Dong-Soo Kwon: Interaction with a display panel – An evaluation of surface-transmitted haptic feedback – Control, Automation and Systems, 2008. ICCAS 2008 International Conference – Pages 278-283 – Oktober 2008

Koert van Mensvoort – What You See Is What You Feel – Designing Interactive Systems 2002 – Pages 345-348 – Juni 2002

Fumihito AMI, Naoya IWATA, and Toshio FUKUDA – Transparent Tactile Feeling Device for Touch-Screen Interface – Robot and Human Interactive Communication, 2004. ROMAN 2004 – Pages 527-532 – September 2004

Hiroshi Ishii: Tangible Bits: Beyond Pixels – Tangible and embedded interaction TEI'08 – 11 Pages – 2008

NXP – LPC2478 Preliminary data sheet – Rev. 01 – 6.July 2007

NXP – UM10237 – LPC24XX User Manual – Rev 04 – 26. August 2009

Trevor Martin: The Insider's Guide To The Philips ARM7-Based Microcontrollers – ISBN: 0-9549988 1 – April 2005

Admatec – NLC800T70D480CTMK29 - Datasheet– Rev. 1 – Mai 2008

Shanghai Tianma Micro-Electronics – TM104SBH01 – Rev 2.1 Dez. 2009

Elite Semiconductor Memory Technology Inc.- M12L2561616A –Datasheet - Revision: 1.5 – JAN 2010

Texas Instruments – SN65HVD231 – Datasheet - SLOS346K – MARCH 2001– Feb. 2011

National Semiconductor – LM1117 – Datasheet - April 2006

NXP - AN10256 – Using IAP for LPC2000 Microcontrollers – Revision 2 – Okt. 2004

NXP – AN10675 - Interfacing 4-wire and 5-wire resistive touchscreens to the LPC247x – Revision 2 – Nov 2008

NXP – AN10771 - EMC peripheral to drive SDRAM – Revision 01 – Dez. 2008

NXP – AN10916 - FAT library EFSL and FatFs port on NXP LPC1700 Rev. 2 — 6 July 2010 Application – July 2010

12 Layout des Prototypen

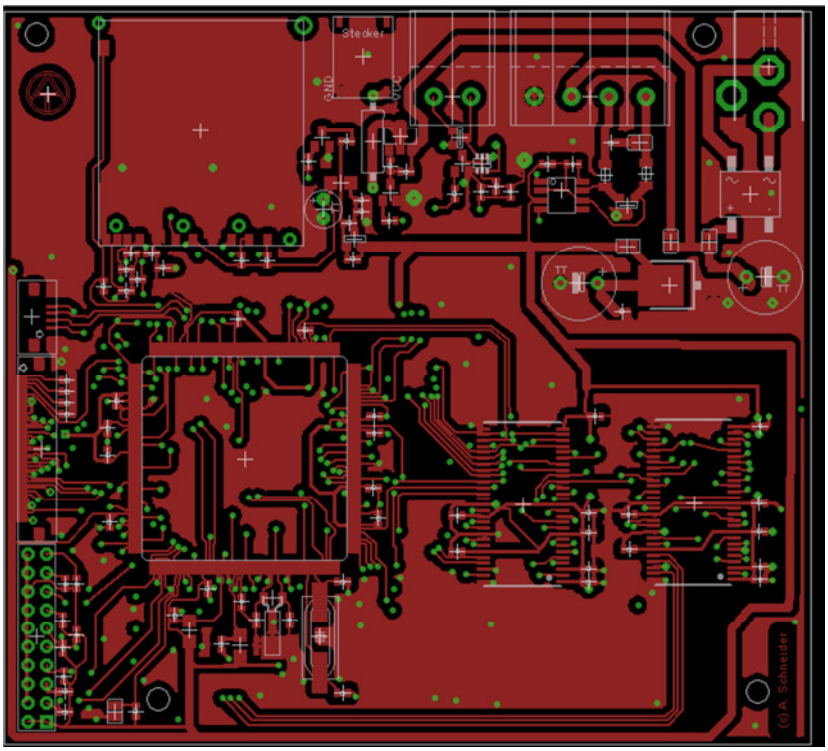

Abbildung 19: Layout Top Layer

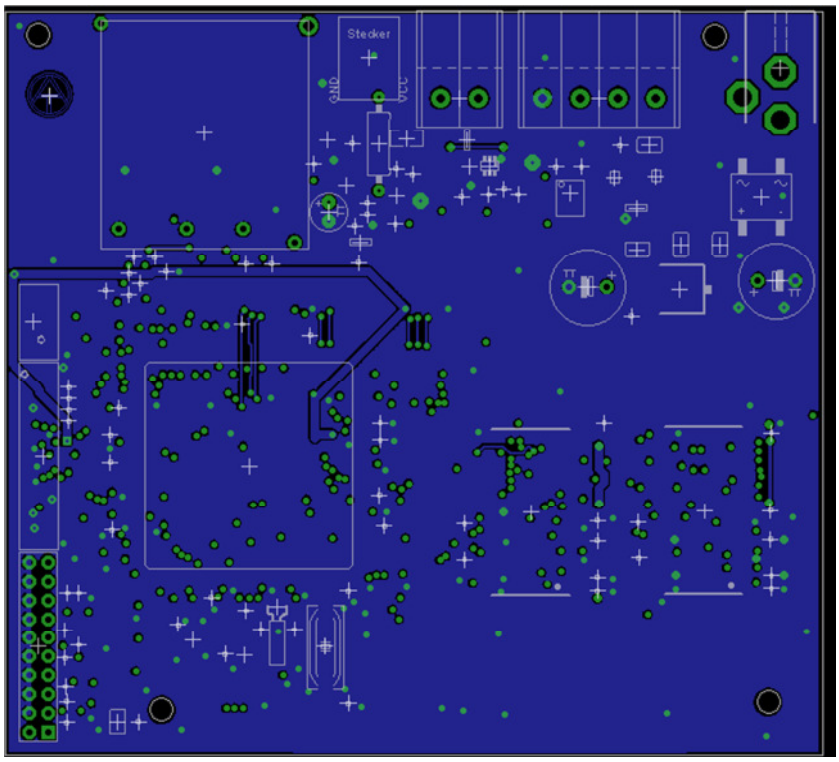

Abbildung 20: Layout Bottom Layer

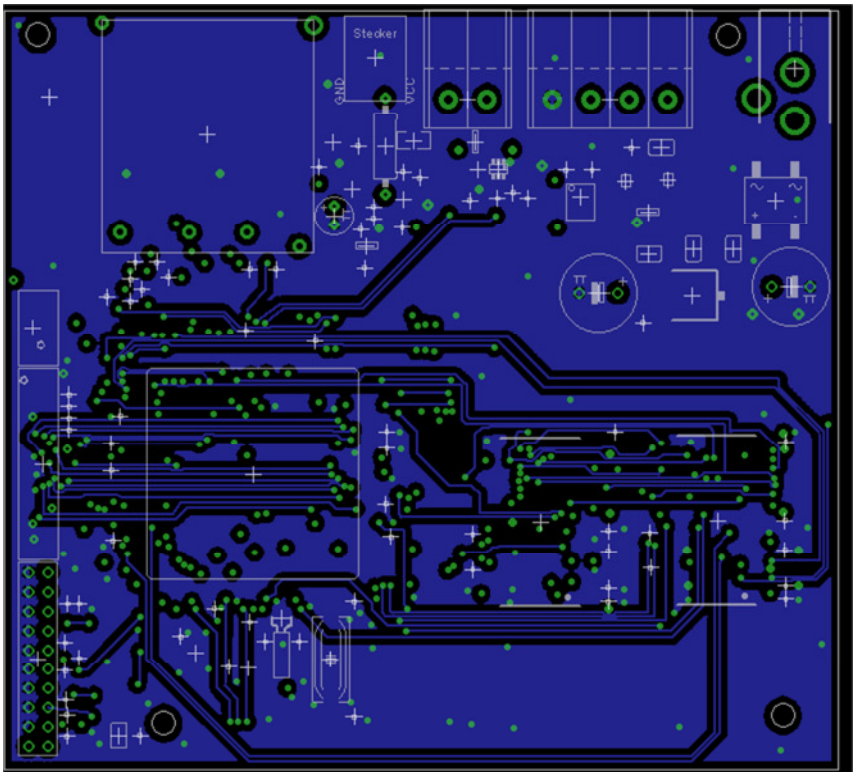

Abbildung 21: Layout Route1 Layer

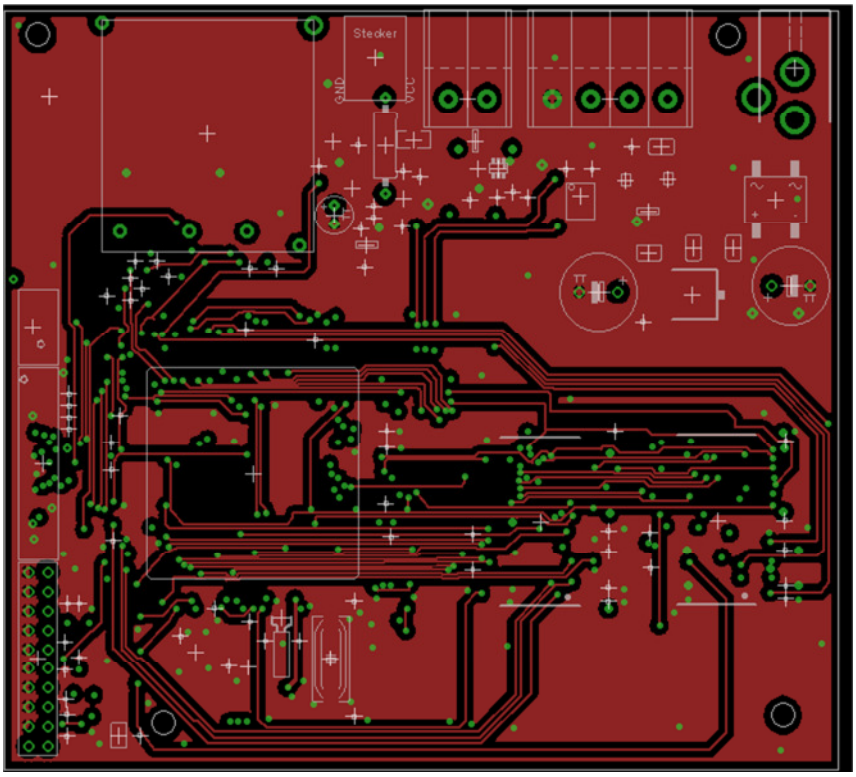

Abbildung 22: Layout Route2 Layer

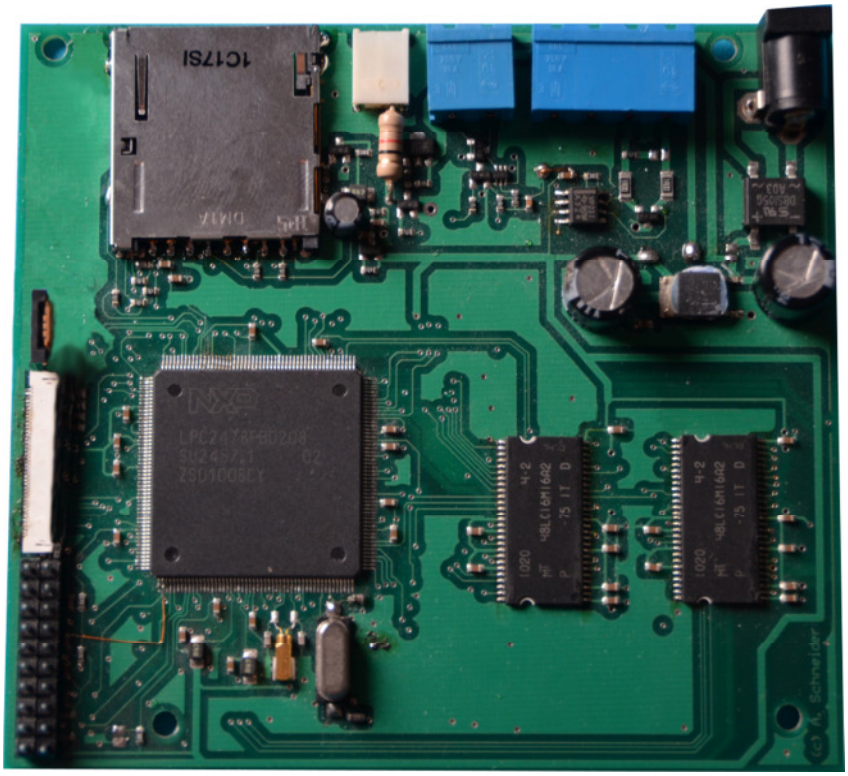

Abbildung 23: Bestückte PCB