Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Torsten Persson

Optische Auswertung von ein- und zweidimensionalen Codes mithilfe des ARM Cortex M3 Mikrocontrollers

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Torsten Persson

Optische Auswertung von ein- und zweidimensionalen Codes mithilfe des ARM Cortex M3 Mikrocontrollers

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Jochen Schneider
Zweitgutachter : Prof. Dr.-Ing. Karl-Ragmar Riemschneider

Abgegeben am 04.Januar 2013

Torsten Persson

Thema der Bachelorthesis

Optische Auswertung von ein- und zweidimensionalen Codes mithilfe des ARM Cortex M3 Mikrocontrollers

Stichworte

Mikrocontroller, Stellaris LM3S9B92, ARM Cortex M3, eindimensionale Codes, Code39, Strichcode, zweidimensionale Codes, QR Code

Kurzzusammenfassung

Diese Arbeit umfasst die Umsetzung der Dekodierung von ein- und zweidimensionalen Codes. Als eindimensionaler Code wurde der Code39, als zweidimensionaler der QR Code gewählt. Der für die Dekodierung verwendete Mikrocontroller ist der ARM Cortex M3, welcher auf dem Stellaris LM3S9B92 Evaluationsboard eingesetzt wird. Die Ausgabe der dekodierten Information wird in einem Terminalprogramm auf einem PC angezeigt.

Torsten Persson

Title of the paper

Optical evaluation of one- and two-dimensional Codes using the ARM Cortex M3 microcontroller

Keywords

Microcontroller, Stellaris LM3S9B92, ARM Cortex M3, one-dimensional code, code39, Barcode, two-dimensional code, QR code

Abstract

Inside this report the implementation of the decoding of one- and two-dimensional codes is described. The code39 is selected as the one-dimensional code, the QR code is chosen as the two-dimensional code. The ARM Cortex M3 which is implemented on the Stellaris LM3S9B92 evaluation kit is used for the decoding. The output of the decoded information is displayed in a terminal software on a PC.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die vorliegende Arbeit befasst sich mit der optischen Auswertung von ein- und zweidimensionalen Codes. Die Umsetzung dieser Auswertung erfolgt mithilfe eines ARM Cortex M3 Mikrocontrollers, welcher auf einem Evaluationskit LM3S9B92 von Texas Instruments eingesetzt wird.

Die Arbeit ist grundsätzlich in zwei Aufgaben unterteilt, die sich durch die verschiedenen Codearten ergeben.

Die eine Aufgabe bezieht sich auf einen bereits bestehenden Laborversuch des Labors für Informationstechnik und beinhaltet das Einlesen und Dekodieren eines eindimensionalen Codes.

Die andere ist das Einlesen und Dekodieren eines zweidimensionalen Codes. Hierbei muss eine Kamera benutzt werden, um den Code zu erkennen und die Daten zu dekodieren. Zweidimensionale Codes haben in vielen Breichen die eindimensionalen ersetzt. Sie haben den Vorteil, dass sie mehr Information auf der gleichen oder teilweise sogar einer kleineren Fläche speichern und trotzdem mit relativ geringen Hardwareanforderungen dekodiert werden können. Durch die heutzutage starke Verbreitung von Mobiltelefonen mit eingebauter Kamera nimmt der Einsatz von zweidimensionalen Codes auch außerhalb des ursprünglichen Einsatzes als Bauteil- oder Produktkennzeichnung zu.

Im Verlauf dieser Arbeit wird zunächst ein allgemeiner Überblick über häufig verwendete ein- und zweidimensionale Codes gegeben und deren Aufbau anhand von Beispielen erklärt. Anschließend werden die technischen Eigenschaften des Evaluationskits aufgeführt und die verwendete Entwicklungsumgebung beschrieben.

Als nächstes folgt die Beschreibung des eindimensionalen Codes. Es wurde der Code 39 ausgewählt, da dieser in dem Laborversuch festgelegt ist. Nach der Beschreibung der für diese Aufgabe eingesetzten Hardware folgt die Schilderung des Programmablaufs.

Anschließend werden der Aufbau und die besonderen Merkmale des zweidimensionalen Codes beschrieben. Es wurde der „Quick Response“ Code (kurz: QR Code[1]) gewählt, da dieser weltweit einer der am meisten verwendeten Codes ist und vielseitig eingesetzt werden kann. Auch hier folgt darauf die Beschreibung der spezifischen Hardware und des Programmablaufes. Am Schluss folgt noch die Festlegung der einzuhaltenden Rahmenbedingungen.

Der vollständige Quellcode, welcher für die Umsetzung der Aufgaben erstellt wurde, befindet sich nur auf der beigelegten CD. Die Programmierung des Mikrocontrollers erfolgte in C.

[1] Der Ausdruck QR Code ist in Japan, den Vereinigten Staaten von Amerika, Australien und Europa ein eingetragenes Warenzeichen der Denso Wave Incorporated.

QR Code is a registered trademark of DENSO WAVE INCORPORATED.

2 Grundlagen

In diesem Kapitel wird ein Überblick über die verschiedenen ein- und zweidimensionalen Codes geliefert und ein Vergleich bezüglich der Informationsdichte und Fehlerkorrekturmöglichkeiten angestellt. Des Weiteren werden das verwendete Mikrocontrollerboard und die Entwicklungsumgebung beschrieben.

2.1 Überblick zu ein- und zweidimensionale Codes

1952 wurde in den USA das erste Patent (Woodland & Silver, 1952) für einen eindimensionalen Code (auch Strichcode genannt) erteilt. Seit dem wurden viele verschiedene Strichcodes eingeführt und einige werden heute noch verwendet. Circa 40 Jahre später wurde der zweidimensionale Code erfunden. Auch hierbei entwickelten sich mehrere Arten. Im Folgenden werden die wichtigsten und am häufigsten verwendeten ein- und zwei- dimensionalen Codes aufgeführt, beschrieben und verglichen.

In den folgenden Kapiteln wird zwischen Strichen (oder auch Balken), Zellen und Zeichen unterschieden. Eindimensionale Codes bestehen aus schwarzen und weißen Strichen, welche jeweils eine logische 1 oder 0 repräsentieren. Bei zweidimensionalen Codes übernehmen die ebenfalls schwarzen und weißen Zellen diese Funktion. Ein Zeichen ist das Ergebnis der Dekodierung von einer unterschiedlichen Anzahl an Stichen bzw. Zellen und ist menschenlesbar.

2.1.1 Eindimensionale Codes

Der Aufbau von eindimensionalen Codes ist bei allen Varianten sehr ähnlich. Er besteht immer aus verschieden breiten, parallel zueinander angeordneten schwarzen Strichen und Lücken (weiße Striche). Oftmals gibt es mehrere Breiten

wobei laut Norm (ISO/IEC 15416) ein Breitenverhältnis von 1 : 2 oder 1 : 3 einzuhalten ist. In einigen Fällen ist die Höhe der Balken ebenfalls genormt.

Der in Deutschland wohl bekannteste Strichcode ist der Global Trade Item Number Code (GTIN-13-Code), welcher früher als European Articel Number Code (EAN-13-Code) beschrieben wurde. Er befindet sich auf nahezu jedem käuflich erwerbbaren Produkt und beinhaltet eine dreistellige Länderbezeichnung, eine vier- bis sechsstellige Betriebsnummer des Herstellers, eine drei- bis fünfstellige Artikelnummer sowie eine Prüfziffer. Anhand dieser Nummer ist ein Produkt weltweit eindeutig erkennbar.

Beispiel:

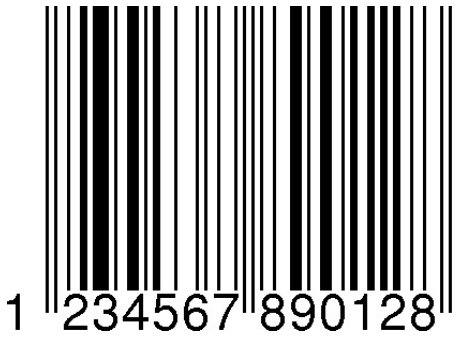


Abbildung 2.1 GTIN-13-Code

Der GTIN-13-Code besteht aus 95 Balken mit gleicher Breite. Die Balken sind entweder schwarz oder weiß und repräsentieren somit entweder eine 1 oder eine 0. Jede Ziffer wird durch sieben Balken dargestellt. Wie in Abbildung 2.1 zu erkennen ist, werden gleichfarbige, aufeinanderfolgende Balken zu einer Linie (max. vier schwarze Balken) bzw. zu einem Freiraum (max. vier weiße Balken) zusammengefasst. Die Information ist infolgedessen mithilfe der Linienbreite kodiert. Um das Dekodieren zu vereinfachen steht am Anfang und am Ende von jedem Code die Folge 101 und in der Mitte steht immer 01010. Der Zeichensatz umfasst ausschließlich die Zahlen 0 - 9.

Ein weiterer häufig genutzter Strichcode ist der Zielcode der deutschen Post. Dieser wird auf jede Briefsendung und Postkarte aufgebracht und dient zur schnelleren maschinellen Sortierung. Der Code besteht aus 5mm hohen Strichen, welche in einem 15mm hohen und 150mm langen Bereich am rechten unteren Rand einer Briefsendung aufgebracht werden (siehe Abbildung 2.2).

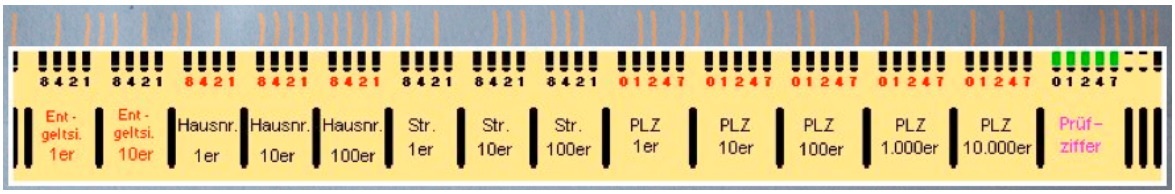


Abbildung 2.2 Zielcode mit Dekodierschablone

Wie in der Abbildung 2.2 sehen ist, beinhaltet der Code die Hausnummer, Strasse, Postleitzahl und eine Prüfziffer. Die Kodierung erfolgt dabei in 4- oder 5-Bit Segmenten. Innerhalb eines Segments zählt jeder aufgedruckte Strich als 0 und die unbedruckten Stellen als 1.

Es gibt noch viele weitere Strichcodes wie beispielsweise:

- Codebar

Abbildung 2.3 Beispiel Codebar

- Code 128

Abbildung 2.4 Beispiel Code 128

- ITF (Interleaved two of five) Code

Abbildung 2.5 Beispiel ITF Code

- Code 39

Abbildung 2.6 Beispiel Code 39

Diese lassen sich jedoch nicht auf eine spezielle Anwendung reduzieren, sondern werden häufig in speziellen Branchen oder Systemen verwendet, oder sind inzwischen von zweidimensionalen Codes ersetzt worden. Zu beachten sind auch die unterschiedlichen Längen der Codes, obwohl alle die gleiche Ziffernfolge repräsentieren. Die ist durch die verschiedenen Codierungsverfahren zu erklären.

Der in Abbildung 2.6 gezeigte und in dieser Arbeit verwendete Code 39 wird ebenfalls noch in einigen Branchen eingesetzt und wird in Kapitel 3.1 näher beschrieben.

2.1.2 Zweidimensionale Codes

Im Gegensatz zu den eindimensionalen Codes gibt es viele verschiedene Möglichkeiten zweidimensionale Codes aufzubauen.

Im Allgemeinen lassen sich zweidimensionale Codes in vier verschiedene Arten unterteilen:

- Stapelcodes
- Matrixcodes
- Punktcodes
- Composite Codes

Der Vorteil dieser Codes ist, dass die Information nicht nur ein einer Richtung kodiert werden, sondern sich auf eine Fläche verteilen. Dadurch lassen sich im Vergleich zu Strichcodes wesentlich mehr Daten auf der gleichen Fläche speichern.

Die vier verschiedenen Arten werden im Folgenden jeweils anhand eines Beispiels näher betrachtet.

Stapelcodes:

Diese Art der Kodierung sieht einem eindimensionalen Code noch sehr ähnlich: „Das Grundprinzip der zweidimensionalen Staplecodes besteht darin, dass mehrere eindimensionale Codes übereinander angeordnet werden, was das Gesamtgebilde der zweidimensionalen Variante ergibt.“ (Lenk, 2002, S. 30)

Als Beispiel ist der Codablock F Code gewählt, da dieser einer der ersten zweidimensionalen Codes war und das Prinzip gut veranschaulicht.

Abbildung 2.7 Beispiel Codablock F

In Abbildung 2.7 ist gut zu erkennen, dass der Codablock F Code aus 2 übereinanderliegenden Code 128 Strichcodes besteht, die ein gemeinsames Start- und Stoppzeichen haben. Die Anzahl der Zeilen kann zwischen 2 und 44 betragen. Die Dekodierung erfolgt zeilenweise.

Weitere wichtige Stapelcodes sind beispielsweise der Code 49 und der PDF (Portable Data File) 417 Code, die nicht auf einen bestehenden Strichcode aufbauen.

Matrixcodes:

Ein Matrixcode hat keine Ähnlichkeit mehr mit einem Strichcode: „Hier sind es schwarze oder weiße Matrixelemente bzw. Zellen, die je nach Position innerhalb einer definierten Matrixstruktur die Daten codieren.“ (Lenk, 2002, S. 15) Durch die Platzierung der Codeelemente in einer Matrix kann ein solcher Code in jeder beliebigen Lage eingelesen werden. Dazu braucht es jedoch einige Elemente, die keine Daten beinhalten, sondern als Lageerkennung und zur Generierung eines Referenzgitters dienen. Durch Fehlerkorrekturberechnungen können diese Codes schlechte Lesbarkeit, Verschmutzungen oder Beschädigungen ausgleichen.

Abbildung 2.8 Beispiel Data Matrix

Abbildung 2.8 zeigt einen Data Matrix Code in der Größe 16x16. Dieser wird vor allem in der Teilekennzeichnung aber auch als elektronische Briefmarke eingesetzt. Es ist zu erkennen, dass die linke und die untere Kante nur aus schwarzen Zellen bestehen während die obere und die rechte Kante immer abwechselnd schwarz und weiß ist. Dies ist die Referenzlage und dient der oben angesprochenen Lageerkennung und Referenzgitter Generierung. Die Zellen innerhalb dieses Rahmens enthalten die Daten. Je mehr Daten gespeichert werden sollen, desto größer wird der Code. Maximal kann eine Größe von 144x144 Zellen erreicht werden.

Andere oft verwendete Matrixcodes sind der Aztec Code (online Ticket der Deutschen Bahn) sowie der in dieser Arbeit benutzte QR Code, der in Kapitel 4.1 genau beschrieben wird. Der Data Matrix und Aztec Code sind Entwicklungen aus den USA und wurden auch vorwiegend dort und in Europa eingesetzt. Der QR Code kommt aus Japan und war zuerst nur im asiatischen Raum verbreitet. Inzwischen wird der QR Code weltweit eingesetzt und hat die anderen Arten teilweise ersetzt. Das Erscheinungsbild der Codes ist sehr ähnlich, was sich auf eine ähnliche Funktionsweise zurückführen lässt.

Punktcodes:

Diese Art von Code ist eine Sonderform des Matrixcodes. Anstelle von weißen und schwarzen Zellen werden in einer festen Matrix Punkte gesetzt oder frei gelassen. Der hier als Beispiel gewählte Dot Code A wird vor allem in der Kennzeichnung von mechanischen Teilen und Baugruppen verwendet, da er sehr klein ist und sich auch mithilfe von Druck-, Stanz- oder Bohrverfahren einfach aufbringen lässt. Dies macht ihn besonders robust gegenüber äußeren Einflüssen und Beschädigungen.

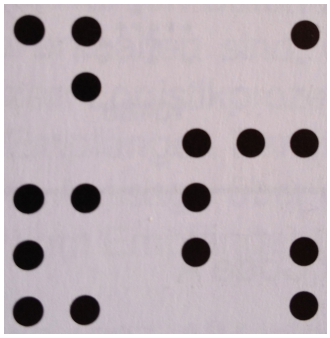


Abbildung 2.9 Beispiel Dot Code A (Lenk, 2002, S. 395)

Abbildung 2.9 zeigt einen Dot Code A mit der kleinstmöglichen Matrix von 6x6. Maximal kann die Matrix 12x12 betragen. Auch die Punktcodes haben feste Bestandteile um die Lage zu erkennen. Bei diesem Code sind zehn Punkte fest vorgegeben um die Referenzlage zu definieren. Die übrigen Punkte beinhalten die Daten, wobei üblicherweise ein gesetzter Punkt einer 1 und eine freie Stelle einer 0 entsprechen.

Composite Codes:

Ein Composite- oder auch Doppelcode ist die Zusammenführung eines Strichcodes mit einem zweidimensionalen Code. Dabei werden häufig bereits vorhandene Codes benutzt und vereinigt. Der Vorteil ist, dass sich mit herkömmlichen Strichcodelesern immer noch ein Teil der Information lesen lässt und trotzdem Zusatzinformationen geliefert werden können. Eingesetzt wird dies zum Beispiel bei Lebensmitteln. Die Artikelnummer befindet sich im Strichcode und Information zu dem Verfallsdatum im zweidimensionalen Code. Das gewählte Beispiel ist für genau diesen Zweck gedacht.

Abbildung 2.10 Beispiel EAN 13 mit CC-B

Der in Abbildung 2.10 gezeigte Code ist der EAN13 mit Composite Code -B (CC-B). Das bedeutet der Strichcode entspricht dem in Kapitel 2.1.1 aufgeführten EAN 13 Code. Der zweidimensionale Anteil mit der Bezeichnung CC-B beinhaltet einen Micro PDF 417 Code der zu den Stapelcodes gehört.

Je nach Speicherbedarf lassen sich viele verschiede Composite Codes erstellen. Häufig wird neben den EAN mit CC auch der Aztec Mesas verwendet, da dieser nicht auf einen bestimmten Strichcode festgelegt ist.

2.2 Überblick Hardware und Software

Der in dieser Arbeit eingesetzte Mikrocontroller und die Entwicklungsumgebung wurden der Einrichtung des Labors für Informationstechnik an der HAW Hamburg nachempfunden. Diese besteht im Wesentlichen aus dem Evaluationskit Stellaris LM3S9B92 und dem Code Composer Studio v5 von Texas Instruments. Das Evaluation Kit besteht aus zwei Platinen, dem LM3S9B92 Board und dem In-Circuit Debug Interface Board (BD-ICDI).

2.2.1 LM3S9B92 Evaluation Kit

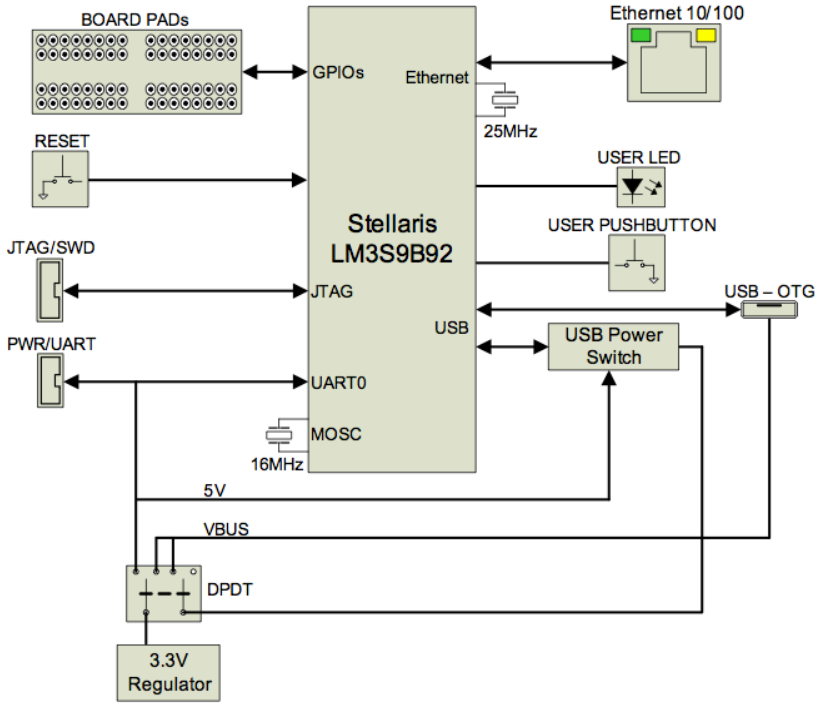


Abbildung 2.11 schematisches Blockdiagramm des LM3S9B92 aus (Texas Instruments, 2011)

Die schematische Darstellung des Stellaris LM3S9B92 Board (Abbildung 2.12) zeigt die vorhandene Hardware. In Abbildung 2.12 ist durch farbliche Markierungen aufgezeigt, wo sich die jeweilige Hardware auf dem Board befindet.

- 10/100 MBit Ethernet Anschluss (rot)

- Full Speed (12Mbits) USB 2.0 OTG[2] Anschluss (braun)
- Große Anschlussflächen zum Kontaktieren der Pins des Mikrocontrollers (grau)
- Frei programmierbarer Taster und LED (blau)
- 16 MHz Quarz für den Systemtakt (orange)
- Einen acht- und einen zehnpoligen Anschluss zum Verbinden mit dem BD-ICDI (gelb)
- Ein Reset-Taster (magenta)

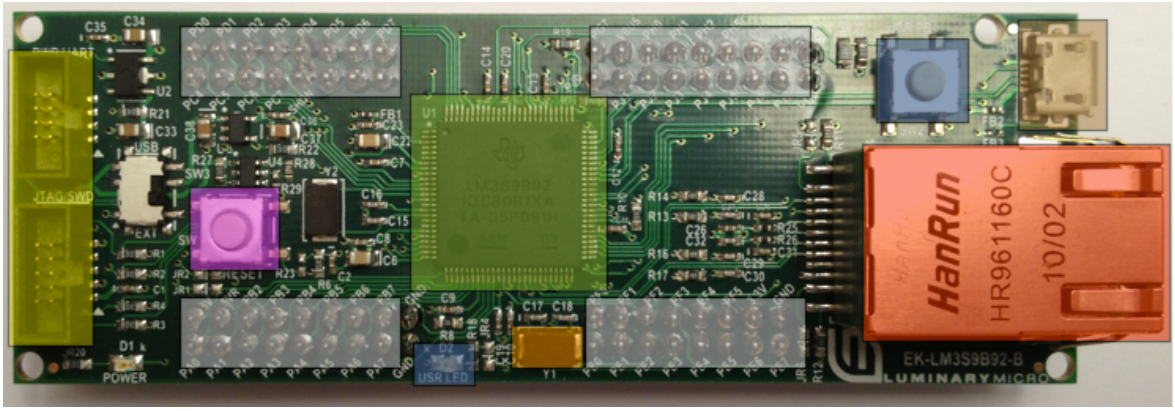


Abbildung 2.12 LM3S9B92 mit farblich markierter Hardware

Das In-Circuit Debug Interface Board ist die zweite eigenständige Platine. Eine schematische Darstellung dieser Platine ist in Abbildung 2.13 verzeichnet.

[2] OTG: on the go, Kommunikation zwischen Geräten, ohne Hostfunktionalität eines Computers

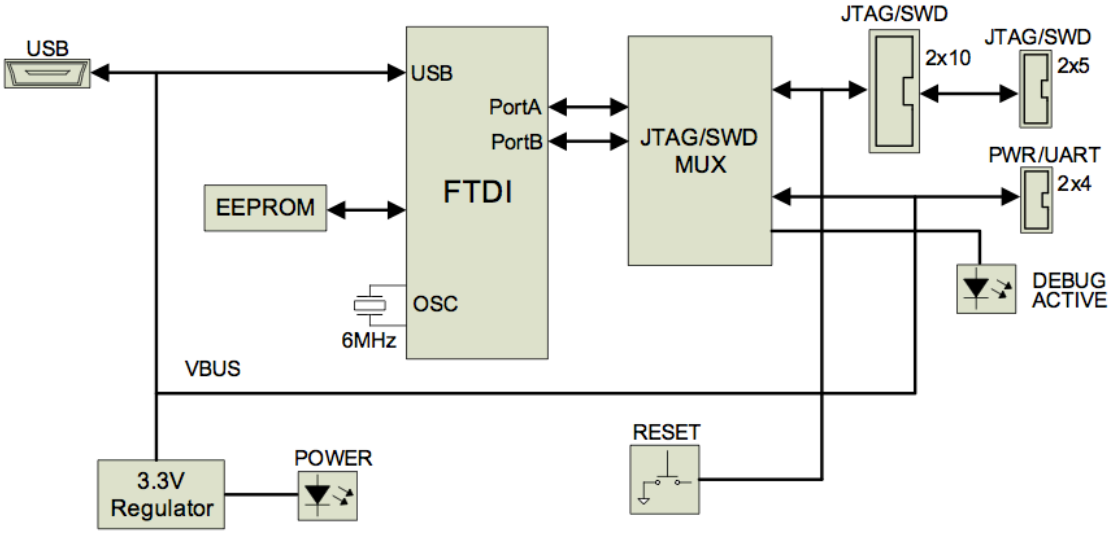


Abbildung 2.13 schematisches Blockdiagramm des In-Circiut Debug Interface Board

Die wichtigsten Hardwarebausteine sind im Folgenden angegeben und in Abbildung 2.14 farblich markiert.

- USB Anschluss zum Verbinden mit einem PC (violett)
- Ebenfalls einen acht- und einen zehnpoligen Anschluss zum Anschließen des LM3S9B92 Boards (gelb)
- „Power on“ LED und „Debug Active“ LED (weiß)
- FTDI FT2232 Chip[3] (grün)

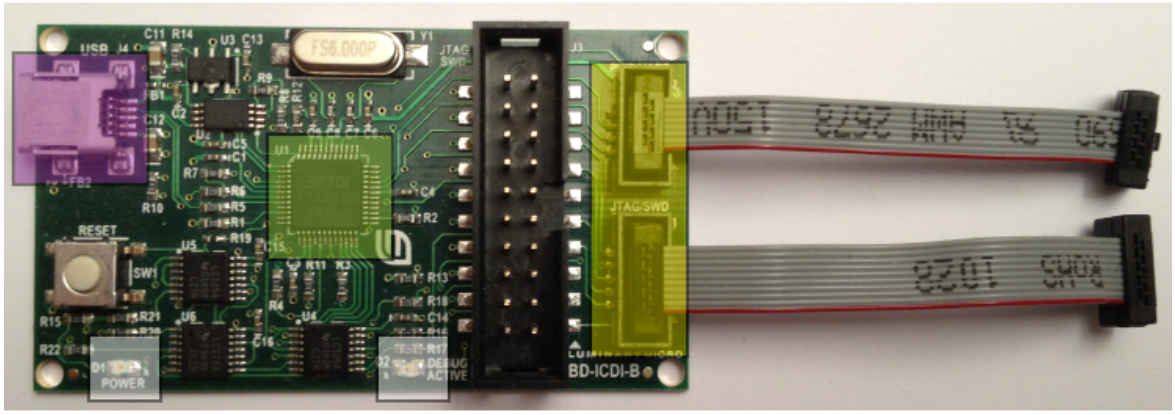


Abbildung 2.14 In-Circuit Debug Board

[3] FTDI FT2232: Future Technology Devices International, konvertiert RS-232 Signale zu USB

Über das BD-ICDI Board kann das LM3S9B92 Board mit der Entwicklungsumgebung auf dem PC verbunden werden. Dafür wird eine JTAG/SWD[4] Verbindung aufgebaut, welche ein komfortables Debuggen[5] ermöglicht. Außerdem wird die Spannungsversorgung an das LM3S9B92 Board angeschlossen und eine UART[6]-Verbindung zwischen dem Mikrocontroller und dem PC aufgebaut. Die achtpolige Leitung verbindet die Spannungsversorgung und die UART-Verbindung, die zehnpolige Leitung schließt die JTAG/SWD Schnittstelle an.

Das BD-ICDI Board wird über USB angeschlossen, der PC erkennt jedoch auch einen COM Port. Diese virtuelle serielle Schnittstelle wird durch einen „FTDI FT2232“ Chip auf dem BD-ICDI Board ermöglicht, welcher die USB Kommunikation auf dem Board in eine dem RS-232 Standard entsprechende umwandelt und somit eine Kommunikation über die UART-Verbindung ermöglicht.

Diese Verbindung wird benutzt, um Ausgaben von dem Mikrocontroller in einem Terminalprogramm auf dem PC anzuzeigen. Dafür werden zwei Leitungen benötigt, eine zum Senden (Transmit, TX) und eine zum Empfangen (Receive, RX). Die Baudrate beträgt 115200 Symbole/Sekunde. Eine zusätzliche Taktleitung ist nicht notwendig, da die übertragenen Datenpakete eine feste Länge haben und sich die Teilnehmer darüber synchronisieren können.

Die 65 GPIO[7] Pins der acht Ports[8] des Mikrocontrollers werden auf dem LM3S9B92 Board nach außen geführt. Dadurch kann externe Hardware einfach angeschlossen werden. Die Anzahl der Pins pro Port ist unterschiedlich:

---

[4] JTAG/SWD: Join Test Action Group/Serial Wire Debug: Verfahren um einen in eine Arbeitsumgebung integrierten Mikrocontroller zu testen und den Programmablauf nachzuvollziehen

[5] Debuggen: Testen des Programmablaufs auf dem Mikrocontroller

[6] UART: Universal Asynchronous Receiver Transmitter: serielles Übertragen von Daten in beide Richtungen

[7] GPIO: General Purpose In Out, multifunktionale Ein- und Ausgabeverbindungen

[8] Ports: mehrere GPIO Pins werden zu einem Port zusammengefasst und bekommen eine gemeinsame Bezeichnung

- Port A (PA): acht Pins (0-7)
- Port B (PB): acht Pins (0-7)
- Port C (PC): acht Pins (0-7)
- Port D (PD): acht Pins (0-7)
- Port E (PE): acht Pins (0-7)
- Port F (PF): sechs Pins (0-5)
- Port G (PG): drei Pins (0,1,7)
- Port H (PH): acht Pins (0-7)
- Port J (PJ): acht Pins (0-7)

Die LED ist mit Pin 0 an Port D (PD0) verbunden und kann über diesen angesprochen werden. Der Taster ist an Port B, Pin 4 (PB4) angeschlossen und nicht entprellt. Diese Verbindungen können über die Jumper JR3 bzw. JR4 gelöst werden.

Für die oben beschriebene UART-Verbindung mit dem PC werden die Pins PA0 (RX) und PA1 (TX) benötigt, auch wenn keine Steckverbindung an diese angeschlossen werden muss.

Es ist zu beachten, dass die GPIO Pins für verschiedene Aufgaben verschiedene Funktionen haben können, jedoch immer nur für eine dieser Aufgaben konfiguriert sind. PA0, PA1, PD0 und PB4 können daher nicht mehr für andere Zwecke eingeplant werden, wenn die oben beschriebenen Funktionalitäten benötigt werden.

Das Hauptmerkmal des Stellaris LM3S9B92 Board ist der Mikrokontroller LM3S9B92, welcher einen 32-Bit ARM[9] Cortex M3 Kern besitzt. Die wesentlichen Merkmale sind dem Datenblatt (Texas Instruments, 2012) entnommen und lauten wie folgt:

- High Performance: 80-MHz operation
- 256 KB single-cycle Flash memory

[9] ARM: Advanced RISC Machines, entwickeln Mikroprozessordesigns, welche von Lizenznehmern gefertigt werden

- 96 KB single-cycle SRAM
- Internal ROM loaded with StellarisWare
- External Peripheral Interface (EPI)
- Advanced Communication Interfaces: UART, SSI, I2C, I2S, CAN, Ethernet MAC and PHY, USB
- System Integration: general-purpose timers, watchdog timers, DMA, general-purpose I/Os
- Analog support: analog and digital comparators, Analog-to-Digital Converters (ADC), on-chip voltage regulator
- JTAG and ARM Serial Wire Debug (SWD)

Weitere wichtige Merkmale sind die Möglichkeiten an jedem GPIO Pin einen externe Interrupt[10] auslösen zu können sowie mehrere UART-Verbindungen gleichzeitig zu betreiben.

Die für die beiden Aufgaben spezifischen Hardwareeigenschaften werden in dem jeweiligen Kapitel erläutert.

2.2.2 Entwicklungsumgebung

Die zum Programmieren und Debuggen benutzte Software ist das Code Composer Studio v5 (Version 5.1.1.00028) von Texas Instruments. Diese bietet die Möglichkeit für jede Aufgabe ein Projekt zu erstellen und in diesem die jeweiligen C- und Headerdateien zu verwalten. Es übernimmt außerdem das Compilieren, Linken und lädt den ausführbaren Quellcode in das SRAM[11] des Mikrocontrollers. Im Debugmodus können die einzelnen Register, die erstellten Variablen und der dazugehörige Speicherbereich angezeigt werden. Der Programmablauf kann in Einzelschritten oder bis zu einem vorher eingefügten Haltepunkte ausgeführt werden.

[10] Interrupt: kurze Unterbrechung des Hauptprogramms um eine Eingabe zu verarbeiten

[11] SRAM: Static Random Access Memory: statischer Speicher mit wahlfreiem Zugriff

Es ist außerdem möglich eine Terminalfunktion in den Debugmodus zu integrieren. Diese muss nachträglich hinzugefügt werden. Eine detaillierte Beschreibung dafür kann im Internet unter: „How to install the terminal plugin in CCSv5“ (o.V., 2012) nachgelesen werden.

Als eigenständiges Terminalprogramm wurde „HTerm“ benutzt. Dies ist eine frei im Internet erhältliche Software, die genaue Einstellungen der gewünschten Verbindung erlaubt und die empfangenden Daten als Logfile speichern kann. Sie ist unter: „Der-Hammer.info“ (Hammer, 2008) zu finden.

Bei der Erstellung des Programmcodes wurden außerdem die StellarisWare Bibliotheken benutzt. Diese enthalten vorgefertigte Funktionen, welche das Konfigurieren der Peripherie vereinfacht. Beim Aufruf einer dieser Funktionen wird durch die Übergabeparameter festgelegt, wie die gewählte Peripherie eingestellt werden soll. Ein weiterer Vorteil ist, dass diese Funktionen im ROM des Mikrocontrollers gespeichert sind und von dort aus ausgeführt werden. Es wird somit weniger Platz im SRAM verbraucht.

Die vollständige Einrichtung der Entwicklungsumgebung wurde so vorgenommen, dass die erstellten Projekte direkt auf einem Arbeitsplatz im Labor für Informationstechnik lauffähig sind.

3 Eindimensionalen Code auswerten

Dieser Teil der Arbeit beschäftigt sich mit dem Einlesen und Auswerten eines eindimensionalen Codes. Als Grundlage der hier bearbeiten Aufgabe dient ein Laborversuch mit dem Titel „Ansteuerung eines Barcode-Lesestift“ (Schneider, 2004), welcher für das Fach Mikroprozessortechnik entwickelt wurde. Die Anforderung ist es, den für das Hitachi H8S/2357 System entworfenen Versuch auf dem Stellaris LM3S9B92 Board umzusetzen.

Grundsätzlich werden bei diesem Versuch der Barcode Lesestift P 51 von Datalogic, welcher in Kapitel 3.2 beschrieben wird, an den Mikrocontroller angeschlossen, die von dem Stift übermittelten Daten ausgewertet und auf dem PC zur Anzeige gebracht. Das genaue Vorgehen wird in Kapitel 3.3 beschrieben. Es wurde der Code 39 gewählt, da er einen übersichtlichen Aufbau und eine gut durchschaubare Decodierung aufweist. Ein Beispiel eines Code 39 wurde in Kapitel 2.1.1 bereits gezeigt. Das folgende Kapitel erläutert die Standardisierung, den Verwendungszweck und den Aufbau genauer.

3.1 Code 39

Der Code 39 wurde 1973 von der Firma Intermec in Everett, Washington, USA erfunden und war der erste alphanumerische Barcode. Noch heute ist er aufgrund des einfachen Aufbaus einer der meistverbreiteten Codes in der Industrie. Vor allem in der Automobil- und Pharmaindustrie wird er nach wie vor auf Versandetiketten und Arzneimittelverpackungen eingesetzt. 1999 wurde er erstmals in der ISO/IEC[12] 16388 spezifiziert. Im Jahre 2007 folge die zweite Version der Spezifikation.

[12] ISO: Internationale Organisation für Normung
IEC: Internationale Elektrotechnische Kommission

Die Bezeichnung Code 39 geht auf die Struktur der Kodierung zurück. Diese setzt sich aus neun Balken pro Zeichen zusammen, welche zwei unterschiedliche Breiten aufweisen und abwechselnd schwarz und weiß sind. Drei der neun Balken müssen breit sein. Das Verhältnis von schmalen zu breiten Balken muss dabei zwischen 1 : 2 und 1 : 3 liegen. Die weiße Lücke zwischen zwei Zeichen trägt keine Information. Das Start- und Endzeichen eines Codewortes ist der Asterisk (*).

0	1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	„Space“	-	.	$
/	+	%							

Tabelle 1 Zeichensatz des Code 39

Der Zeichensatz des Code 39 entspricht dem alphanumerischen Zeichensatz plus sieben Sonderzeichen (Tabelle 1). Der Asterisk dient nur als Start- und Endzeichen und ist somit nicht im Zeichensatz enthalten.

Beispiel:

Abbildung 3.1 Code 39 mit Zeichentrennung

Im Gegensatz dazu gibt es noch den erweiterten Code 39 welcher den kompletten ASCII[13] Zeichensatz darstellen kann. Dadurch nimmt jedoch die Informationsdichte

[13] ASCII: American Standard Code for Information Interchange.
Sieben Bit Zeichenkodierung: 2^7 =128 Zeichen

ab, da zwei Zeichen für ein Klarschriftzeichen verwendet werden müssen. Diese Erweiterung wurde im Rahmen dieser Arbeit nicht umgesetzt.

3.2 Hardwareaufbau

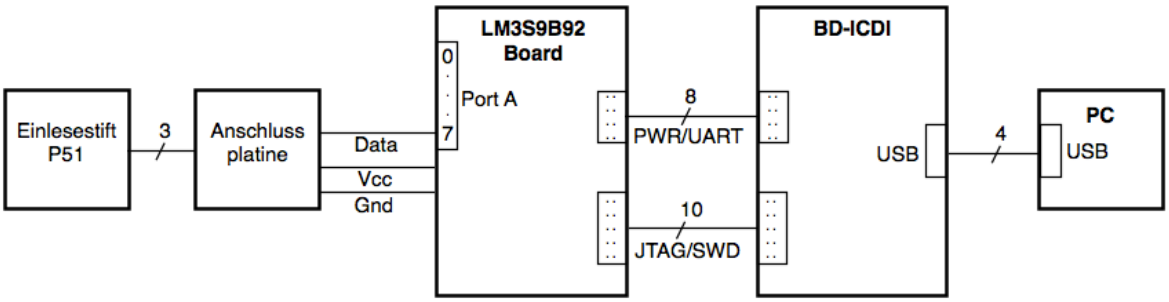


Abbildung 3.2 Hardwareaufbau zum Einlesen eines Code39

Das Stellaris LM3S9B92 Board ist über USB mit dem PC verbunden und bezieht somit auch die Versorgungsspannung daher. Wie bereits in Kapitel 2.2.1 beschrieben besteht eine UART-Verbindung zwischen dem Board und dem PC (Abbildung 3.2). Der Lesestift hat einen neunpolige D-Sub Stecker (Abbildung 3.3) welcher sich nicht direkt an das Stellaris LM3S9B92 Board anschließen lässt. Es wird eine zusätzliche Platine eingesetzt (Abbildung 3.4), welche die 5V Spannungsversorgung von dem Stellaris Board an den Lesestift legt und die Datenleitung an Port A Pin 7 anschließt. Die Datenleitung wird zusätzlich auf der Platine über eine „Pullup“ Widerstand auf die Versorgungsspannung gezogen.

Abbildung 3.3 Lesestift P51 von Datalogic mit D-Sub Stecker

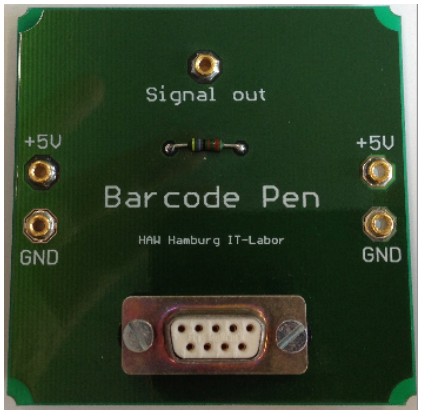


Abbildung 3.4 Anschlussplatine für den P51 an den Mikrocontroller

Die technischen Daten des P51 sind:

Betriebsspannung:	+5 V DC
Lichtquelle:	rote LED mit $\lambda = 600\,\text{nm}$
Tiefenschärfe:	1,5 mm
Lesegeschwindigkeit:	bei 0,15 mm Modulbreite zwischen 5 cm/s und 1 m/s
	bei 0,25 mm Modulbreite zwischen 7 cm/s und 2,4 m/s
Lesewinkel:	max. 50°
Datenausgang:	Low bei Lesespitze auf weiß
Signalflanken:	5 µs Anstieg und Abfall
Fremdlichteinfluß:	max. 4000 lux

Abbildung 3.5 Technische Daten P51 aus (Schneider, 2004)

Beim Überstreichen des Barcodes wird ein dynamisches Ausgangssignal erzeugt, welches bei schwarzen Balken +5 V (high, binär: 1) und bei weißen Balken 0 V (low, binär: 0) beträgt. Verweilt der Lesestift für mehr als 60 ms auf einem schwarzen Balken, wird das Ausgangssignal auf low gezogen.

Als Beispiel ist der Spannungsverlauf für das Strichmuster von *0* in Abbildung 3.6 zu sehen.

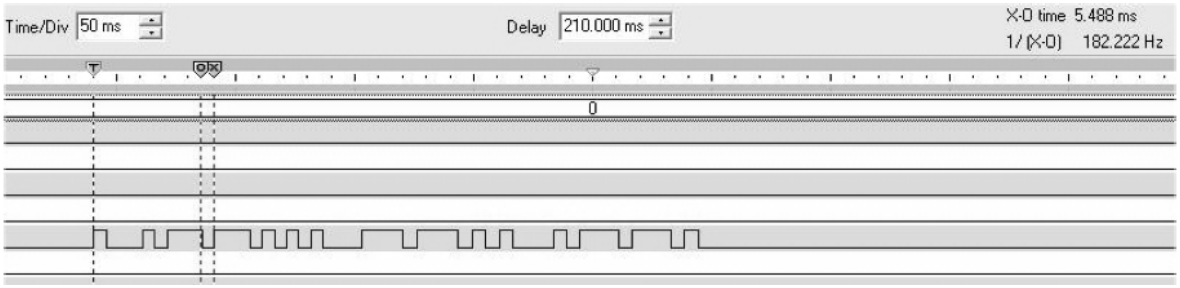


Abbildung 3.6 Spannungsverlauf am Ausgang des P51 aus (Schneider, 2004)

Um den Inhalt des eingelesenen Codes anzuzeigen, ist das Stellaris LM3S9B92 Board über USB als virtueller COM Port an den PC angeschlossen und kann somit mit einem Terminalprogramm kommunizieren. Die dekodierten Zeichen können dadurch in Klartext auf dem Bildschirm angezeigt werden.

3.3 Umsetzung auf dem Mikrokontroller

Der Informationsgehalt eines Code 39 ist mithilfe der Balkenbreiten kodiert. Beim Dekodieren muss diese Information somit gemessen und bewertet werden. Der folgende Programmablaufplan (Abbildung 3.7) zeigt wie dies auf dem Stellaris LM3S9B92 Board umgesetzt wurde.

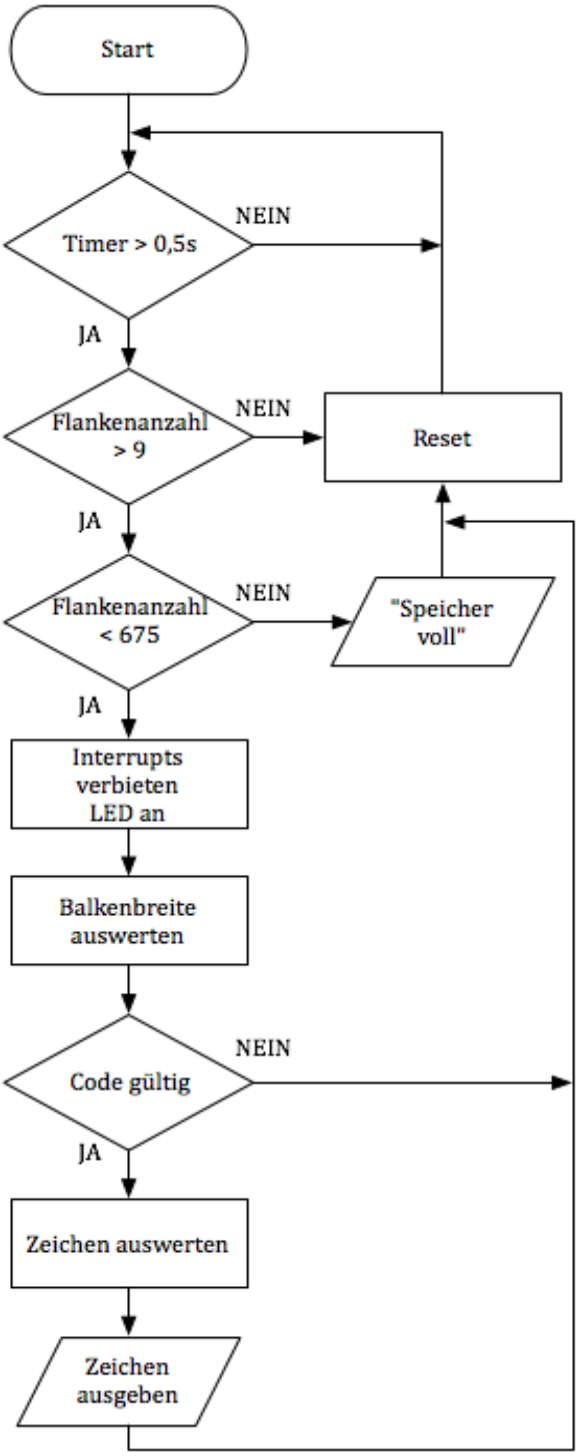


Abbildung 3.7 Programmablauf zum Dekodieren eines Code 39

Der komplette Quellcode zu der dem Programmablauf in Abbildung 3.7 entsprechenden Dekodierung eines Code 39 befindet sich im Anhang A. Wie oben beschrieben, gibt der Lesestift beim Überstreichen eines Barcodes eine dynamische Spannung von 0V oder 5V aus. Der Informationsgehalt steckt jedoch nicht in der Amplitude, sondern die Breite der Balken ist entscheidend. Der Mikrocontroller muss also erkennen wie breit ein Balken ist und wann der nächste Balken anfängt. Dafür wird ein externer Interrupt verwendet, der ausgelöst wird, wenn ein Flankenwechsel auf der Datenleitung erkannt wird. Die Erkennung der Balkenbreite geschieht in der Interrupt Service Routine („balkenbreite()“) mithilfe eines Zählers. Der Ablauf der Interrupt Service Routine (ISR) geschieht nach dem in Abbildung 3.8 aufgezeigtem Programmablaufplan.

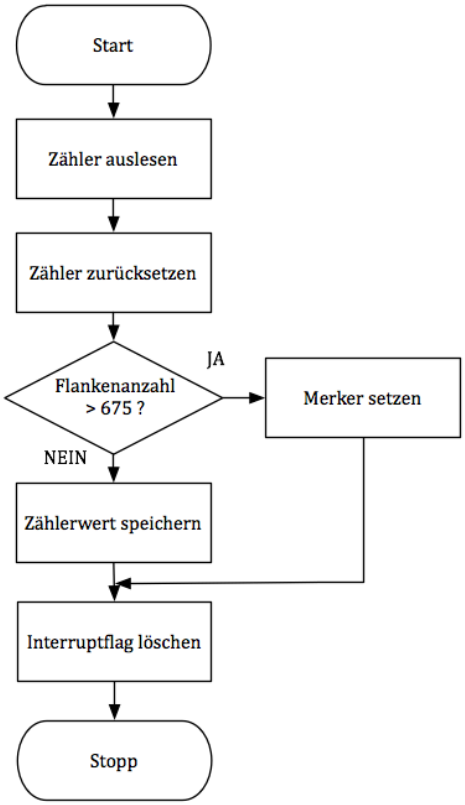


Abbildung 3.8 Programmablauf ISR

Wird der Interrupt ausgelöst, wird zunächst der Wert des Zählers ausgelesen. Danach wird überprüft ob die maximal Anzahl der erlaubten Flanken erreicht ist. Ist dies der Fall, wird ein Marker gesetzt und die Funktion beendet. Wenn die Anzahl

noch nicht erreicht ist wird der Wert des Zählers in einem Array gespeichert. Danach wird der Zähler wieder zurückgesetzt. Der nächste Flankenwechsel löst den Interrupt erneut aus und der Wert des Zählers wird in dem nächsten Arrayelement gespeichert. Die Anzahl der Flankenwechsel wird ebenfalls gespeichert. Beispielhaft zeigt die Abbildung 3.9 Werte, die beim Einlesen des * Zeichens entstanden sind. Die binäre Darstellung dieses Zeichens lautet: 010010100.

Expression	Type	Value
timer_value	int[200]	0x2000026C
[0 ... 99]		
[0]	int	538937
[1]	int	41352
[2]	int	81834
[3]	int	38072
[4]	int	38795
[5]	int	80608
[6]	int	35144
[7]	int	77156
[8]	int	32317
[9]	int	33455
[10]	int	31360

Abbildung 3.9 Array mit gespeicherten Zählerwerten

Auffällig ist, dass elf Werte aufgenommen wurden, obwohl ein Codewort nur aus neun Balken besteht. Der erste und letzte Wert wird bei der Dekodierung nicht berücksichtigt. Der erste Wert entsteht bei der ersten steigenden Flanke und entspricht somit noch keiner Balkenbreite. Der letzte Wert entspricht der weißen Lücke, welche nach jeweils neun Blaken eingefügt wird. Nachdem das letzte Zeichen des Codes eingelesen wurde, entstehen durch das Abheben des Stiftes ebenfalls noch Flanken, die ignoriert werden. Relevant sind nur die neun Werte in den Arrayelementen [1] bis [9]. Anhand dieser Werte ist auch erkennbar, dass die breiten Balken ungefähr zweimal so breit sind wie die schmalen.

Wenn ein Code eingelesen wird und 500ms lang keine neue Flanke kommt, wird davon ausgegangen, dass der Code komplett eingelesen wurde. Nach der letzten Flanke läuft der Zähler weiter. Dieser Wert wird im Hauptprogramm abgefragt. Der

Zähler zählt mit dem Systemtakt hoch. Der Zählerwert für 500ms berechnet sich somit nach folgender Formel:

$$Wert = Takt/Hz * Zeit/s$$

$$Wert = 16 * 10^6 \frac{1}{s} * 0{,}5s = 8 * 10^6$$

Formel 1 Berechnung des Zählerwerts

Die Dekodierung startet erst, wenn der Wert größer als $8*10^6$ ist.

Es gibt zwei Stufen der Dekodierung. Erst werden die Balkenbreiten ausgewertet und als Binärzahl in einem weiteren Array gespeichert, dann werden die ausgewerteten Balken zusammengefasst und dem jeweiligen Klartextzeichen zugeordnet.

Die erste Stufe ist in der Funktion „decode_to_array()“ umgesetzt. Für die Entscheidung ob der gespeicherte Zählerwert eines Balkens einem breiten oder schmalen Balken entspricht, wird für jedes Zeichen (neun Balken) der durchschnittliche Wert bestimmt. Die Werte der neun Balken werden mit dem jeweiligen Durchschnittswert verglichen. Wenn der Wert größer ist, entspricht es einem breiten Balken, ist er kleiner, einem schmalen Balken. Sollte der Wert genau dem Durchschnitt entsprechen wird er ebenfalls als schmaler Balken gewertet da die Wahrscheinlichkeit dafür höher ist. Die neun binären Werte eines Zeichens werden mithilfe der Bitshift Operation in einem Array als short[14] gespeichert (Abbildung 3.10).

[14] Datentyp short = 16 bit = erforderlich um binäre Darstellung von neun Balken zu speichern

Expression	Type	Value
code	short[2...	0x2000058C
[0]	short	0000000010010100
[1]	short	0000000000010110
[2]	short	0000000001010010
[3]	short	0000000110000100
[4]	short	0000000111000000
[5]	short	0000000001000110
[6]	short	0000000100100001
[7]	short	0000000001100001
[8]	short	0000000010100010
[9]	short	0000000100100001
[10]	short	0000000101100000
[11]	short	0000000011000100
[12]	short	0000000010010100

Abbildung 3.10 Zeichen in Binärdarstellung

Bevor die zweite Decodierungsstufe ausgeführt wird, wird überprüft, ob ein gültiger Code eingelesen wurde. Dafür ist die Funktion „check_code()“ zuständig. Diese überprüft, ob das erste und letzte Zeichen der Asterisk ist. Ist die Überprüfung positiv, wird die Freigabe der nächsten Decodierungsstufe gesetzt. Das Hauptprogramm überprüft die Freigabe und beendet die Decodierung mit der Ausgabe „*Fehler*“, wenn sie nicht gesetzt ist. Bei einem gültigen Code wird die Funktion „decode_to_char()“ ausgeführt. Die in dem Array gespeicherten Binärwerte der Zeichen können auch als Dezimalwerte ausgelesen werden (Abbildung 3.11).

Expression	Type	Value
code	short[2...	0x2000058C
[0]	short	148
[1]	short	22
[2]	short	82
[3]	short	388
[4]	short	448
[5]	short	70
[6]	short	289
[7]	short	97
[8]	short	162
[9]	short	289
[10]	short	352
[11]	short	196
[12]	short	148

Abbildung 3.11 Zeichen in Dezimaldarstellung

Diese Dezimalwerte werden in einer „Switch-Case“ Anweisung ausgewertet, dem jeweiligen lesbaren Zeichen zugeordnet und über die UART-Verbindung im Terminal auf dem PC ausgegeben. Die in Abbildung 3.12 gezeigte Ausgabe entsteht durch die in Abbildung 3.11 dargestellten Werte.

```
Serial: (COM11, 115200, 8, 1, None, None - CONNECTED)
TP.WS12/13
```

Abbildung 3.12 Ausgabe im Terminal

Damit während der Dekodierung keine Fehler entstehen, ist der Interrupt in dieser Zeit nicht aktiv. Außerdem leuchtet die LED D2 auf dem LM3S9B92 Board. Erst nachdem die LED erloschen ist, kann eine neuer Code eingelesen werden. Die maximale Codelänge ist auf 75 Zeichen beschränkt. Das entspricht 675 Balken, was bei der kleinsten unterstützten Balkenbreit (0,15mm/Balken) ca. 10,1 cm Codelänge bedeutet. Wird dieses Limit überschritten wird „*Speicher voll*“ im Terminal ausgegeben und die Dekodierung abgebrochen.

4 Zweidimensionalen Code auswerten

Dieser Teil der Arbeit befasst sich mit dem Einlesen und Auswerten von QR Codes. Grundsätzlich muss der Code von einer Kamera aufgenommen, auf dem Bild erkannt und dann dekodiert werden. In den folgenden Unterkapiteln werden die Standardisierung und Verwendung sowie die verschiedenen Versionen und der Aufbau von QR Codes erklärt. Außerdem wird anhand des Ablaufplans gezeigt, wie die Decodierung auf dem Mikrocontroller umgesetzt wurde. Zuletzt werden noch verschiedene Testfälle betrachtet, welche die Rahmenbedingungen festlegen, unter denen ein Einlesen und Dekodieren möglich ist.

4.1 QR Code

Der QR Code wurde im Jahre 1994 von der japanischen Firma Denso Wave entwickelt. Der Begriff „QR Code“ ist in den USA, Japan, Australien und Europa ein eingetragenes Warenzeichen und muss entsprechend gekennzeichnet werden (Fußnote 1, Seite 1). Dieser Hinweis ist bei der Verwendung einer QR Code Abbildung nicht notwendig.

International standardisiert wurde der Code erstmalig 1997 von AIM (Association for Automatic Identification and Mobility). Im Jahre 2000 folgte die ISO/IEC 18004:2000 Standardisierung, welche später zurückgezogen und 2006 durch die ISO/IEC 18004:2006 ersetzt wurde, da neue Modelle entwickelt wurden.

Die Abbildung 4.1 zeigt einen QR Code, welcher den Text „Torsten Persson“ codiert. Es ist zu erkennen, dass der Code aus verschiedenen schwarzen und weißen Flächen besteht. Diese setzen sich aus den sogenannten Zellen zusammen. Eine schwarze Zelle repräsentiert dabei eine logische Eins und eine weiße Zelle eine logische Null. Der genaue Aufbau wird in Kapitel 4.1.2 erläutert.

Abbildung 4.1 QR Code "Torsten Persson"

Ursprünglich wurde der Code für die Kennzeichnung von Baugruppen und Komponenten in der Automobilindustrie entwickelt. Heutzutage finden sich QR Codes in vielen verschiedenen Bereichen auch außerhalb von Logistikanwendungen. Durch die weite Verbreitung von Smartphones und dem mobilen Internet[15] nutzt vor allem die Werbebranche diese Technik. Dafür werden QR Codes auf Plakatwände, in Werbeprospekten oder Schaufestern platziert und können so von Kunden mit einem Smartphone abfotografiert werden. Dem Code wird dabei meist eine Internetadresse hinterlegt, welche den Nutzer direkt auf eine bestimmte Website leitet sobald er den Code eingescannt hat. Allerdings werden auch Visiten-, Eintritts- oder Fahrkarten sowie Hinweisschilder z.B. in Museen mit QR Codes bedruckt, um einen schnellen Informationsaustausch zu gewährleisten oder weiterführende Informationen zur Verfügung zu stellen.

Gerade in Verbindung mit dem (mobilen) Internet entstehen aber auch Gefahren bei dem Umgang mit QR Codes. Dem Code kann nicht angesehen werden, ob die hinterlegte Information vertrauenswürdig ist oder nicht. So können beispielsweise schadhafte Links verbreitet werden, welche beim Öffnen unerwünschte Software installieren. Dies ist allerdings kein Fehlverhalten des Codes und kann auch durch ihn nicht erkannt oder verhindert werden. Es ist Aufgabe der Anwendung, mit welcher der QR Code decodiert wird ein solches Verhalten zu implementieren. Wenn ein Link

[15] Mobiles Internet: Internet Zugang auf einem Mobilfunkgerät über das Mobilfunknetz

zu einer Website decodiert wird, sollte die Anwendung diese nicht ungefragt öffnen oder einen automatischen Download starten. (vgl. Biermann, 2011)

Die Darstellungsgröße eines gedruckten QR Codes ist nicht eindeutig beschränkt und hängt vor allem von dem eingesetzten Lesegerät ab. Der Code muss von einem Lesegerät formatfüllend und differenzierbar erfasst werden können. Es wurden schon Häuserfassaden aber auch Briefmarken mit QR Codes bedruckt.

4.1.1 Versionen

Die ursprüngliche Standardisierung beinhaltete nur ein Modell. In der neuen sind dagegen vier verschiedene Modelle aufgeführt. Es wird dort unterschieden zwischen:

- Model 1
- Model 2
- Model 2005
- Micro QR Code

Das „Model 1“ entspricht dabei der ersten Standardisierung. „Model 2“ ist eine überarbeitete Form des „Model 1“. Hauptsächlich wurden die „timing patterns“ (siehe Kapitel 4.1.2) hinzugefügt um die Lesbarkeit bei großen Codes zu verbessern. Die Möglichkeit auch gespiegelte und farbinvertierte Codes zu erstellen ist mit dem „Model 2005“ umgesetzt worden, welches sich sonst nicht von „Model 2“ unterscheidet. Deutlich differenzierbar hingegen ist der „Micro QR Code“, welcher von „Model 2005“ abgeleitet ist und einen kleineren Aufbau sowie ein begrenztes Datenvolumen aufweist. In dieser Arbeit wurde die Decodierung des „Model 2“ umgesetzt.

Um den zahlreichen Einsatzmöglichkeiten gerecht zu werden, wird der QR Code „Model 2“ in 40 verschiedene Varianten aufgeteilt. Diese unterscheiden sich hauptsächlich durch die Menge an Informationen, die codiert werden können. Die Erhöhung der Datenmenge wird durch Vergrößerung des Codes erreicht. Die kleinste Variante (Variante 1) hat 21x21 Zellen und es können bis zu 25 alphanumerische Zeichen gespeichert werden. Variante 40 ist mit 177x177 Zellen die größte und kann bis zu 4296 alphanumerische Zeichen enthalten. Jede Variante ist um 4x4 Zellen

größer als die vorherige. Die Anzahl der codierbaren Zeichen ist aber nicht nur von der Variante abhängig, sondern auch von dem verwendeten Zeichensatz und dem gewählten Fehlerkorrekturlevel. Folgende Zeichensätze werden unterstützt:

- Numerisch
- Alphanumerisch
- ISO (Latin und Kana)
- Kanji

Das Fehlerkorrekturlevel gibt an, wie viel Prozent des Codes wiederhergestellt werden können. Die Fehler können durch physikalische Beschädigung (Teile des Codes sind verdeckt oder fehlen) oder durch falsches Einstufen der hell/dunkel Übergänge beim Einlesen verursacht werden. Es ist zu beachten, dass nur Fehler bei den Daten korrigiert werden können. Werden Erkennungsmerkmale (siehe Kapitel 4.1.2) verdeckt oder falsch eingelesen, wird der Code gar nicht erst erkannt. Es wird zwischen vier Fehlerkorrekturlevels unterschieden:

- L (low) = 7%
- M (medium) = 15%
- Q (quartile) = 25%
- H (high) = 30%

Je höher das Fehlerkorrekturlevel ist, desto mehr Daten müssen dafür in den Code integriert werden. Die Menge an codierbarer Information wird somit geringer. Tabelle 2 stellt die Anzahl der Zeichen in den jeweiligen Zeichensätzen in Abhängigkeit zu den Fehlerkorrekturlevels für die Varianten 1 und 40 dar (Auszug aus Lenk, 2002, S. 461 f.)

Variante	Fehler-korrektu-level	Anz. Daten-codeworte	Anz. Zeichen Numerisch	Anz. Zeichen Alpha-numerisch	Anz. Zeichen ISO	Anz. Zeichen Kanji
1	L	19	41	25	17	10
	M	16	34	20	14	8
	Q	13	27	16	11	7
	H	9	17	10	7	4
40	L	2956	7089	4296	2953	1817
	M	2334	5596	3391	2331	1435
	Q	1666	3993	2420	1663	1024
	H	1276	3057	1853	1273	784

Tabelle 2 Abhängigkeit Zeichenanzahl - Fehlerkorrekturlevel

4.1.2 Aufbau

In dieser Arbeit wurde die Erkennung von QR Codes des „Model 2“, Variante 1 umgesetzt. Im Folgenden wird der Aufbau dieser Variante erläutert.

Wie bereits in Kapitel 4.1.1 erwähnt, besteht die Variante 1 des QR Codes aus 21x21 Zellen. Als Beispiel wird der in Abbildung 4.1 bereits gezeigte QR Code mit dem Inhalt „Torsten Persson“ verwendet. Die farbigen Markierungen in Abbildung 4.2 zeigen besondere Merkmale, die im Folgenden erklärt werden.

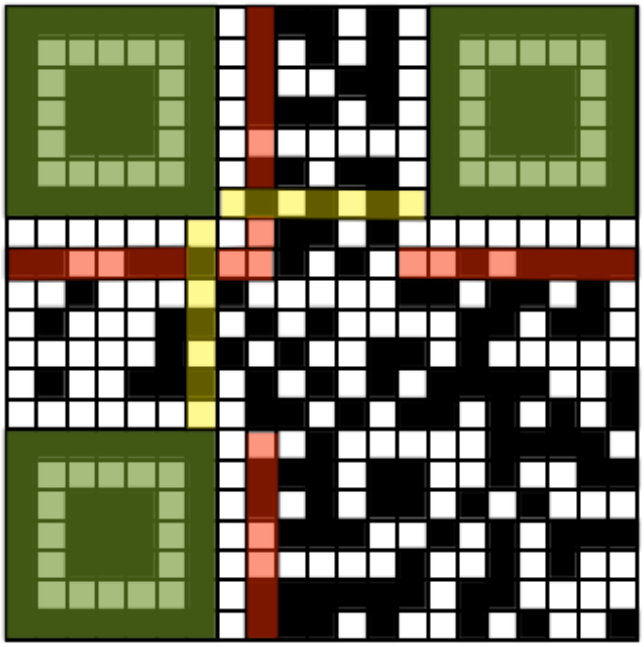


Abbildung 4.2 QR Code mit Gitter und Markierungen

Die drei grün gekennzeichneten Flächen sind die „finder patterns“ oder auch Ecksteine. Sie sind das Haupterkennungsmerkmal eines QR Codes. Die Ecksteine bestehen aus einem 7x7 Zellen großen Rahmen, der eine Zelle breit ist. Das von dem Rahmen eingeschlossene Quadrat hat eine Fläche von 3x3 Zellen und eine Zelle Abstand vom Rahmen (siehe Abbildung 4.3). Das so entstehende 1:1:3:1:1 Muster ist reserviert für die Ecksteine und sollte mit einer hohen Wahrscheinlichkeit ansonsten nicht in dem Code vorkommen.

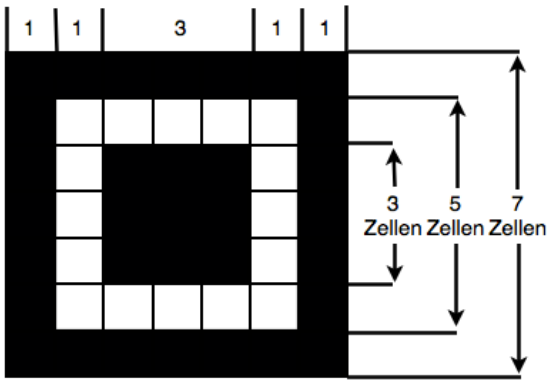


Abbildung 4.3 Aufbau Eckstein

In der Referenzlage befindet sich ein solches Muster immer in den Ecken links oben, links unten und rechts oben. Sie dienen somit der schnellen Erkennung eines Codes

sowie der Lageerkennung. Außerdem lassen sich die physikalische Größe und eine eventuell aufgetretene Verzerrung bestimmen. Wenn ein solches Muster in der rechten unteren Ecke gefunden wird, ist der Code nicht in der Referenzlage und muss entsprechend gedreht werden.

Die Ecksteine sind untereinander mit den „timing pattern“ verbunden. Diese sind in Abbildung 4.2 gelb hinterlegt und verbinden die untere rechte Ecke des Ecksteins in der oberen linken Ecke mit den zwei anderen Ecksteinen. Die „timing pattern“ bestehen aus jeweils abwechselnd schwarzen und weißen Zellen. Da die genaue Position dieser Muster im Code bekannt ist, kann mit deren Hilfe ein Referenzgitter erstellt werden, an welchem wiederum die Koordinaten der Zellen bestimmt werden.

Die in der oben stehenden Abbildung rot markierten Muster beinhalten die Formatinformationen des Codes. Sie bestehen aus 15 Zellen und sind zweimal vorhanden. In Abbildung 4.4 ist die Lage der Zellen nummeriert aufgeführt. Die Bezeichnungen Dx und Ex (x = Platzhalter für eine Nummer) werden weiter unten erklärt.

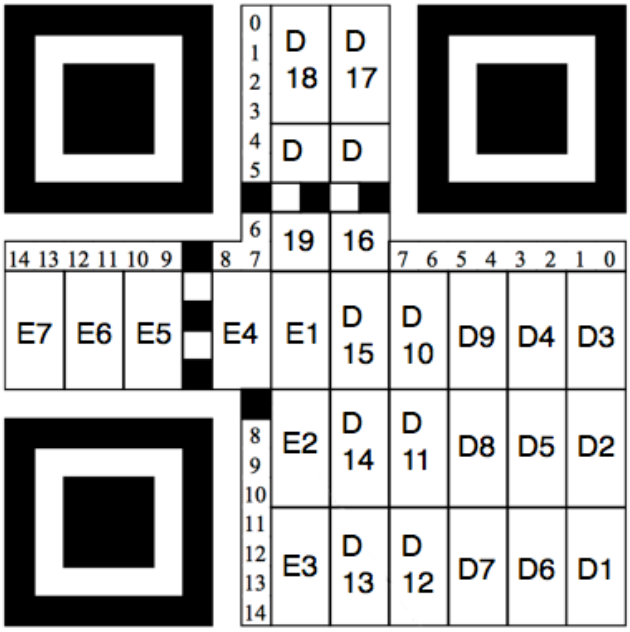


Abbildung 4.4 Aufbau Formatinformation, Daten und EC Symbole

Zehn (5-14) der 15 Zellen dienen der Fehlerkorrektur. Die Zellen 0 und 1 geben Auskunft über das Fehlerkorrekturlevel. Dabei gilt folgende Zuordnung:

Fehlerkorrekturlevel	binäre Darstellung
L	01
M	00
Q	11
H	10

Tabelle 3 Fehlerkorrekturlevel Darstellung in Formatinformation

Die Zellen 2, 3 und 4 geben die Art der Maskierung an. Beim Erstellen eines QR Codes werden die Daten mit einer bestimmten Maske XOR verknüpft um eine möglichst ausgewogene Anzahl von schwarzen und weißen Zellen zu erreichen und um zu verhindern, dass das Muster der Ecksteine auftritt. Dafür stehen acht verschiedene Masken zur Verfügung, die in Abbildung 4.5 grün markiert und mit den entsprechenden binären Darstellungen aufgezeigt sind.

Abbildung 4.5 Maskierungsoptionen

Es ist zu erkennen, dass die Ecksteine, die „timing pattern“ und die Formatinformationen von der Maskierung ausgeschlossen sind. Die angegebene Gleichung gilt für die grün markierten Zellen. Die genaue Vorgehensweise des Maskierens wird an dieser Stelle nicht erläutert, da es für das hier umgesetzte Decodieren nicht von Bedeutung ist. Anhand der in der Formatinformation enthaltenen Nummer (Zellen 2, 3, 4) kann die Maskierung eindeutig erkannt und somit aufgelöst werden.

Die verbleibenden zehn Zellen der Formatinformation werden durch den Bose-Chaudhuri-Hocquenghem[16] (15, 5) Code generiert. Anschließend werden alle 15 Zellen maskiert um sicher zu stellen, dass nicht 15 weiße entstanden sind. Das genaue Verfahren ist nicht relevant, da es für das Dekodieren der Formatinformation eine Tabelle mit allen gültigen Varianten gibt. Die eingelesenen Formatinformationen werden mit diesen gültigen Varianten verglichen, wobei eine Abweichung von maximal drei Zellen zulässig ist. Die Variante, die der eingelesenen Format-information am ähnlichsten ist, wird ausgewählt.

Der in Abbildung 4.2 nicht farblich markierte Bereich beinhaltet die Daten- und Fehlerkorrektur Symbole. Insgesamt gibt es in der Variante 1 26 Symbole, welche sich aus jeweils acht Zellen zusammensetzen. Die Anordnung der Symbole ist in Abbildung 4.4 für das Fehlerkorrekturlevel L dargestellt. Dx steht dabei für ein Datensymbol und Ex für ein „Error Correction“ also ein Fehlerkorrektursymbol. Es ist zu erkennen, dass 19 der 26 Symbole für die Daten verwendet werden. Ein Code in der Variante 1 mit dem Fehlerkorrekturlevel H stellt nur neun Symbole für Daten zur Verfügung. Die restlichen werden für die Fehlerkorrektur benötigt. Die Zellen werden beim Decodieren als Bits mit dem Wert 0 oder 1 interpretiert und so entsteht pro Symbol ein Byte. Die Anordnung der Bits im Symbol ist abhängig von der Platzierung des Symbols. Die in Abbildung 4.4 gezeigten Symbole D1-D3 werden beispielsweise mit dem „Most Significant Bit“ (MSB[17]) zuerst von unten nach oben (Tabelle 4) codiert, während die Symbole D4-D6 mit dem MSB von oben nach unten (Tabelle 5) beschrieben werden.

0	1
2	3
4	5
6	7

Tabelle 4 Anordnung der Zellen von unten nach oben

[16] BCH Code: korrigiert mehrere ein Bit Fehler in einem Datenwort. Benannt nach den drei Entwicklern dieses Codes: R.C. Bose, D. K. Ray-Chaudhuri und A. Hocquenghem

[17] MSB = „Most Significant Bit“ = Bit mit der höchsten Wertigkeit

6	7
4	5
2	3
0	1

Tabelle 5 Anordnung der Zellen von oben nach unten

Auch wenn in dieser Arbeit ausschließlich die Decodierung der oben beschriebene Variante 1 durchgeführt wird, werden der Vollständigkeit halber im Folgenden die Merkmale weiterer Varianten beschrieben. Abbildung 4.6 zeigt einen QR Code in Variante 7 also mit 45x45 Zellen und einem Fehlerkorrekturlevel L, der ebenfalls „Torsten Persson“ enthält. Dieser Code enthält alle oben beschriebenen Merkmale sowie zwei zusätzliche, die farblich markiert wurden.

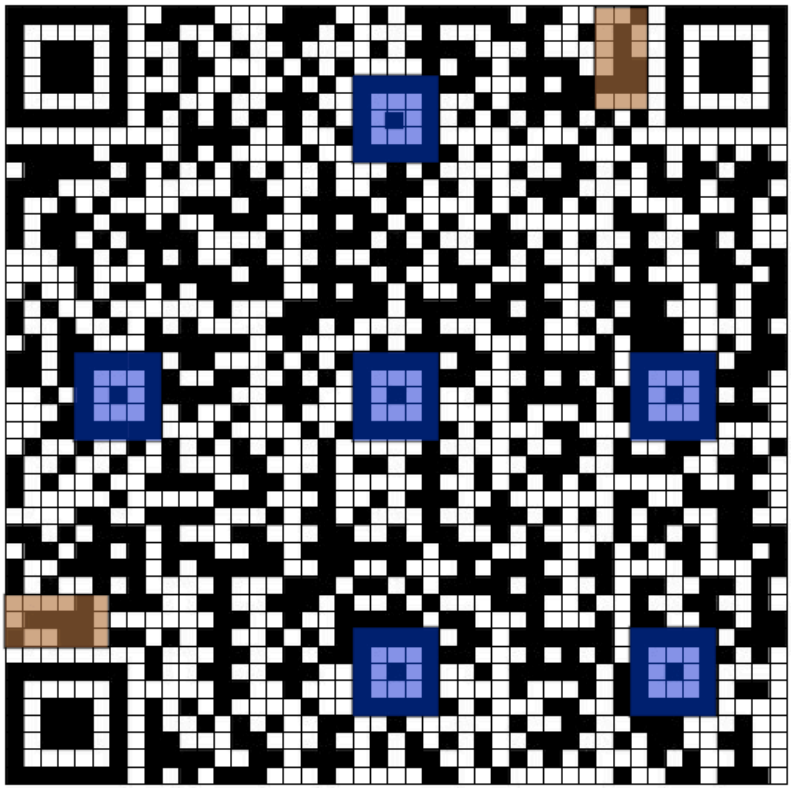


Abbildung 4.6 QR Code Variante 7

Die blau markierten Muster sind die „alignment pattern“, auch Ausrichtungsmuster genannt. Diese sind in allen Varianten außer der Variante 1 enthalten und dienen zur besseren Erkennung des Codes auf dem Bild. Sie bestehen aus einem 5x5 Zellen großen Rand und einer eingeschlossenen Zelle. Wird der Code noch größer,

kommen weitere dieser Muster hinzu. In Variante 40 sind 46 solcher Ausrichtungsmuster enthalten.

Ab Variante 7 wird die Variantennummer ebenfalls angegeben, da sie sich nicht mehr so einfach aus dem Code ableiten lässt. Diese Information besteht aus 18 Zellen, wobei sechs davon die Information tragen und zwölf für die Fehlerkorrektur bestimmt sind. Ähnlich wie bei der Formatinformation ist die Anzeige der Variante auch zweimal im Code enthalten. Die braun markierten Stellen in Abbildung 4.6 zeigen wo sie sich befinden.

4.2 Hardwareaufbau

Zum Einlesen eines zweidimensionalen Codes wird eine Kamera verwendet. Diese liest das Bild ein und sendet es in Graustufen an den Mikrocontroller. Dieser Verarbeitet das Bild, sucht den Code und gibt die in ihm enthaltene Information in einem Terminalprogramm auf dem angeschlossenen Computer aus. In den folgenden Kapiteln werden der Aufbau der Kamera und das Gesamtsystem beschrieben.

4.2.1 Kamera

Die verwendete Kamera ist die CMUcam4. „CMU“ steht dabei für die „Carnegie Mellon University“, an der sie entwickelt wird. Sie wird von Lizenznehmern gebaut und vertrieben. Das vorliegende Modell ist von Lextronic gefertigt. Das eigentliche Kameramodul ist das Omnivision OV9665. Auf dem CMUcam Board wird dieses Modul durch einen eigenen Mikrocontroller (P8X32A von Parallax) und einige Peripherie ergänzt (Abbildung 4.7).

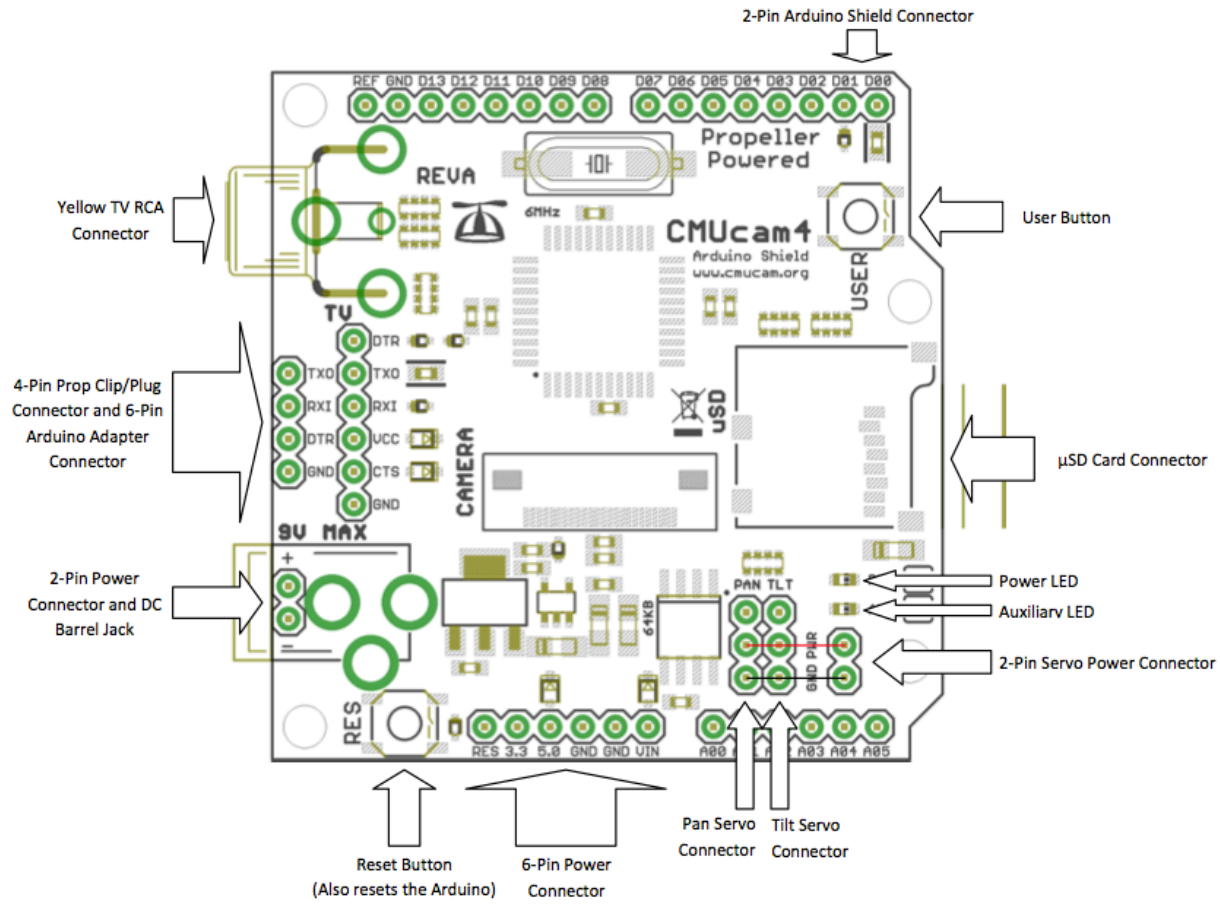


Abbildung 4.7 CMUcam4 Layout und Peripheriebeschreibung aus (Agyeman & Rowe, 2012)

Ein Großteil der gezeigten Peripherie wird für die hier umgesetzte Aufgabe nicht benötigt. Die Kamera wurde ursprünglich entwickelt um Farbverfolgung zu realisieren. Sie wurde ausgewählt, da sie über eine serielle Schnittstelle verfügt und eine einstellbare Auflösung bietet. Außerdem kann sie das aufgenommene Bild in Graustufen umwandeln.

Das dabei verwendete Farbformat ist RGB565. Das bedeutet, dass sich die Farbinformation von jedem Pixel aus 16 Bit zusammensetzen, wobei fünf Bit den Rotwert, sechs Bit den Grünwert und wieder fünf Bit den Blauwert eines Pixels repräsentieren. Bei einem in Grauwerte umgewandelten Bild sind die Werte für rot, grün und blau immer gleich groß. Je höher der Wert ist, desto heller ist die Farbe: alle Bits gleich null repräsentiert somit einen schwarzer Pixel, alle Bits gleich eins einen weißen.

Die Kamera kommuniziert über eine serielle Schnittstelle mit einem Mikrocontroller. Sie empfängt Befehle und sendet die Bilddaten über diese Verbindung. Durch den auf dem CMUcam4 Board vorhandenen Mikrocontroller können die Befehle einfache Strings[18] sein, die auf dem Board zu Anweisungen für das Kameramodul übersetzt werden. Im Folgenden sind die verwendeten Befehle aufgeführt:

- „rs“: Reset, versetzt die Kamera in Ausgangszustand
- „bw 1“: Aktiviert den schwarz-weiß Modus
- „pm 1“: Aktiviert den Pollmodus. Dieser bewirkt, dass die Kamera nur einmal auf eine Anfrage antwortet.
- „sf 2 2“: „send frame“, fordert ein Bild mit der Auflösung 160*120 an.
- „L0“: schaltet die LED aus

Die Kamera ist in der Lage Bilder mit folgenden Auflösungen aufzunehmen: 640, 320, 160, 80 : 480, 240, 120, 60

Die Auflösung von 160*120 wird gewählt, da sie den Kompromiss zwischen Detailtreue und Datenmenge darstellt. Es dauert ca. 15 Sekunden, bis ein Bild vollständig eingelesen und übertragen wurde.

Ein weiterer wichtiger Befehl ist „gh x y“. Er gibt das Histogramm einer Farbe des aktuellen Bildes aus. X steht dabei für die Farbauswahl (0=rot, 1=grün, 2=blau) und y für die Anzahl der Klassen (y=0,1,...,5; Anzahl=2^y), in welche die Farbwerte eingeteilt werden. Mithilfe dieses Befehls kann die Anzahl der Pixel bestimmt werden, die einem bestimmten Farbbereich zugeordnet werden können.

Das in Abbildung 4.7 aufgeführte TV-Verbindungselement kann nicht für eine Echtzeitausgabe des aufzunehmenden Bildes benutzt werden. Eine indirekte Lösung, das aufgenommene Bild anzuzeigen, wird in Kapitel 4.3.2 aufgezeigt.

[18] String: Folge von lesbaren Zeichen.

4.2.2 Gesamtsystem

Wie bereits in Kapitel 2.2.1 erwähnt, besteht das Evaluationskit aus 2 Platinen. Die Spannungsversorgung und Kommunikation erfolgen über den USB Port des angeschlossenen PCs. Die Spannungsversorgung der Kamera erfolgt über das LM3S9B92 Board. Außerdem werden noch eine Sende- und die Empfangsleitung benötigt, um eine UART-Verbindung zwischen dem LM3S9B92 und der Kamera aufzubauen. Diese liegen an Port D Pin 2 (PD2, RX) und Port D Pin 3 (PD3, TX). Die Baudrate dieser Verbindung beträgt 19200 Symbole/Sekunde. Der schematische Hardwareaufbau ist in Abbildung 4.8 dargestellt.

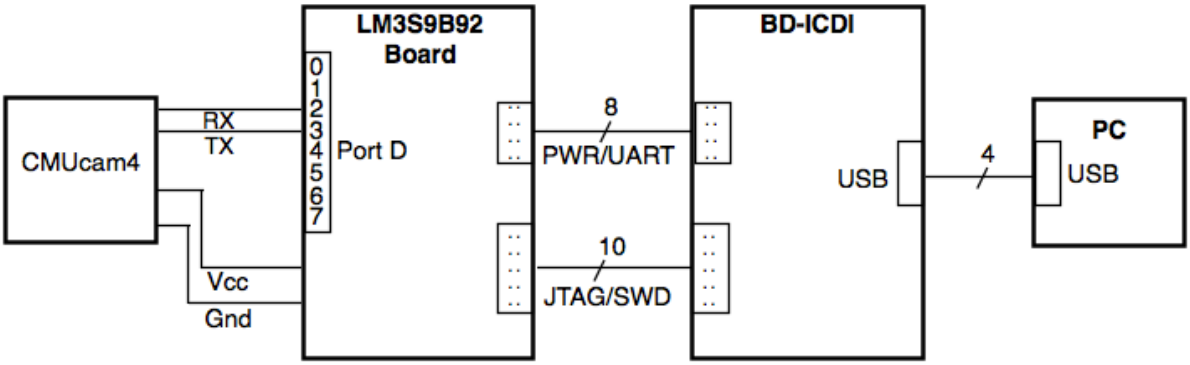


Abbildung 4.8 schematischer Hardwareaufbau zum Dekodieren von QR Codes

4.3 Umsetzung auf dem Mikrocontroller

Zur Erfassung der Bilder und der Dekodierung des Codes ist die Software nach dem in Abbildung 4.9 gezeigten Ablaufplan erstellt worden. Hierbei ist zu beachten, dass die grundsätzliche Struktur und einzelne Funktionen aus einem bestehenden Programm entnommen wurden. Dieses Programm wurde von Daniel Beer als Linuxanwendung erstellt und als „open Source“[19] Lösung im Internet veröffentlicht (Beer, 2012). Das Programm wurde an die Aufgabenstellung und die Gegebenheiten des Stellaris LM3S9B92 Boards angepasst. In dem auf der beigelegten CD angegebenen Quellcode ist bei jeder Funktion ein Kommentar dazu eingefügt, ob die Funktion komplett übernommen, geändert oder neu programmiert wurde. Die im

[19] open Source: Software die frei im Internet erhältlich ist und (abhängig von der Lizenz) frei kopiert, modifiziert und veröffentlicht werden darf.

Anhang ausgeführten Funktionen sind neu programmiert oder stark verändert worden.

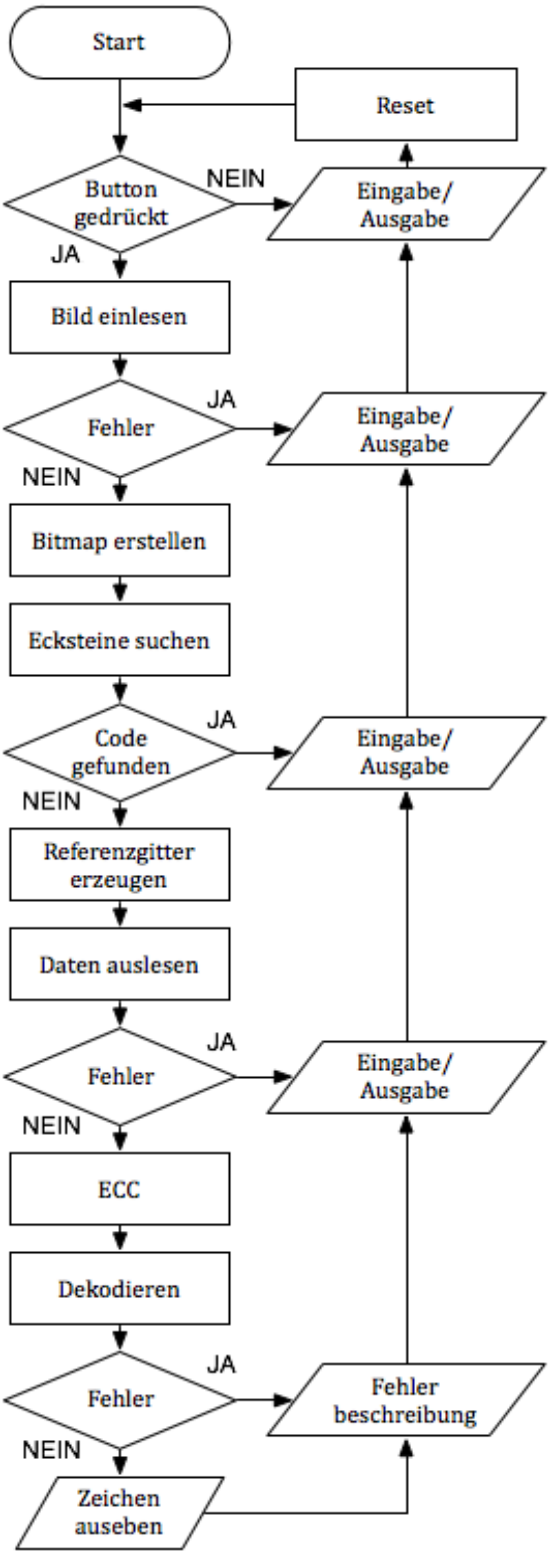


Abbildung 4.9 Flussdiagramm genereller Programmablauf

Wie in Abbildung 4.9 zu sehen ist, wartet der Mikrocontroller auf die Betätigung des User Button welcher sich auf dem Stellaris Board befindet. Das Drücken des Tasters löst einen Interrupt aus, welcher das Startzeichen setzt. Direkt im Anschluss werden die Interrupts deaktiviert damit ein Prellen oder erneutes Drücken des Tasters keine Auswirkungen auf den Programmablauf haben. Die eventuell auftretenden Interrupts werden jedoch automatisch gespeichert und ausgeführt, sobald sie wieder erlaubt werden. Um ein dadurch ausgelöstes Neustarten des Programms nach einem Durchlauf zu verhindern, bleibt das Startzeichen auch nach Beendigung einer Dekodierung noch gesetzt. In der Interrupt Service Routine (start_prog(), Anhang B) wird eine Abfrage des Startzeichens eingebaut. Ist es noch gesetzt, hat die ISR keine Auswirkungen. Am Ende einer Dekodierung werden erst die Interrupts wieder erlaubt und gegebenenfalls abgearbeitet und dann das Startzeichen gelöscht.

Als zentraler Speicherpunkt wird eine Struktur „qr“ angelegt, in welcher alle relevanten Informationen zu dem QR Code gespeichert werden.

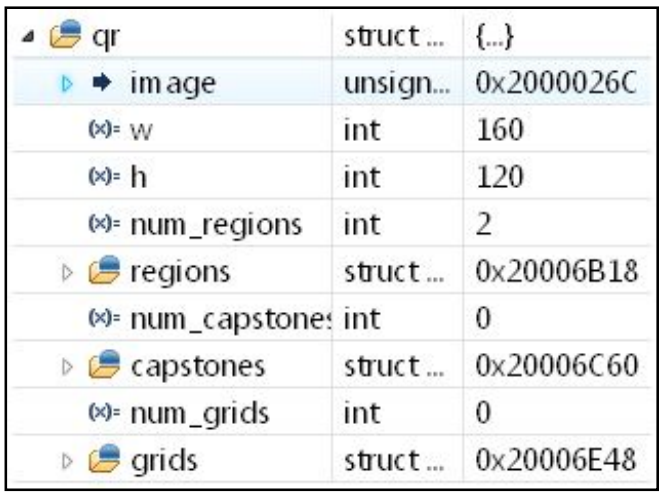


Abbildung 4.10 QR Code Struktur

Abbildung 4.10 zeigt den Inhalt der Struktur nach ihrer Initialisierung. Im Folgenden werden die einzelnen Punkte erklärt:

- Image: beinhaltet die Adresse des Speicherplatzes an der das Bild abgelegt wird
- w, h : geben die Größe des Bildes in Pixel an

- regions: beinhaltet wieder eine Struktur, in welcher gefundene Flächen eingetragen werden (siehe Kapitel 0)
- num_regions: sorgt dafür, dass diese Flächen nicht 0 und 1 genannt werden
- num_capstones: Anzahl der gefundenen Ecksteine
- capstones: enthält eine Struktur, in der die gefundenen Ecksteine eingetragen werden
- num_grids: Anzahl der gefundenen QR Codes
- grids: enthält eine Struktur, in welcher die Informationen für das Referenzgitter und die „timing pattern“ eingetragen wird.

Wie der in Abbildung 4.9 gezeigte Programmablauf auf dem Stellaris LM3S9B92 Board umgesetzt wurde, wird in den folgenden Kapiteln erläutert.

4.3.1 Bild aufnehmen und konvertieren

Abbildung 4.9 zeigt, dass nach dem Drücken des Buttons ein Bild eingelesen wird. Die dafür verwendeten Funktionen befinden sich im Anhang C. Für die Kommunikation mit der Kamera wird ein char[20] Buffer verwendet, welcher 15 ASCII Zeichen aufnehmen kann. Der Befehl für die Kamera wird in dem Buffer gespeichert und dann über die in Kapitel 4.2.2 beschrieben UART-Verbindung verschickt. Die Kamera quittiert den Befehl mit einem String. Dieser wird wieder in dem Buffer gespeichert, wobei der gesendete Befehl überschrieben wird. Anhand dieser Antwort kann überprüft werden, ob der Befehl richtig ausgeführt wurde. Die korrekte Ausführung des Befehlt wird mit dem String „ACK“ bestätigt. Ist in der Kamera ein Fehler aufgetreten wird dies in Abhängigkeit vom Befehl mit dem String „NCK“ oder einem Fehlertext gekennzeichnet. In beiden Fällen wird der Fehler erkannt und eine entsprechende Fehlermeldung im Terminal angezeigt, außerdem wird das Programm unterbrochen und an den Anfangspunkt zurück gesetzt. Abbildung 4.11 zeigt den Ablauf der Kommunikaton zwischen dem Mikrocontrollerboard und der Kamera.

[20] Datentyp char = 8 bit = 1Byte, kann ein ASCII Zeichen speichern

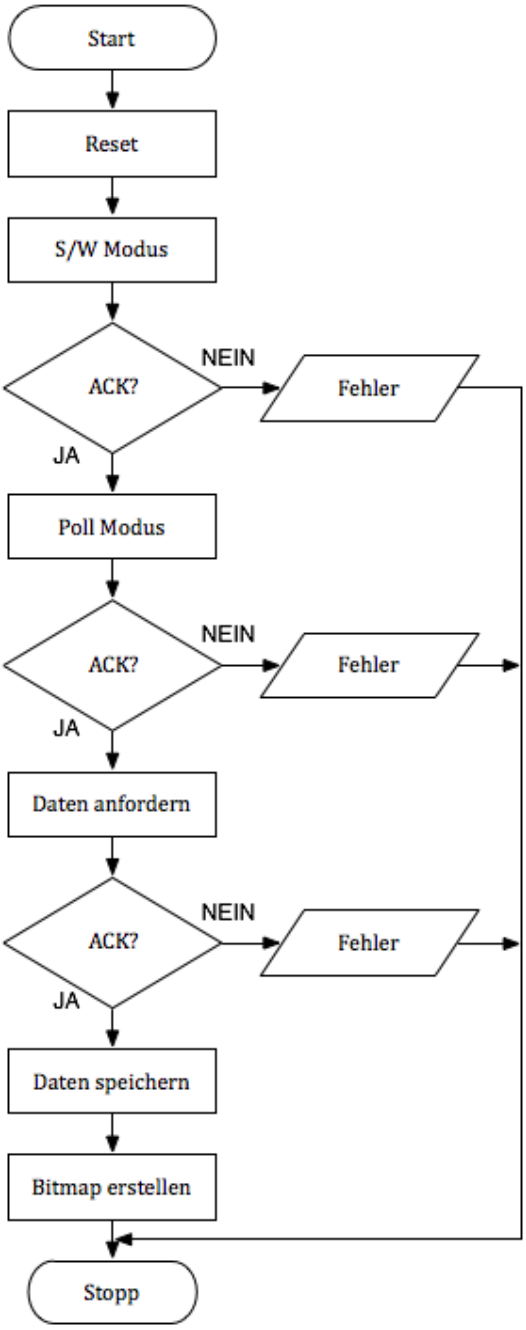


Abbildung 4.11 Ablauf der Kommunikation Board - Kamera

Wie in Abbildung 4.11 schon aufgezeigt ist, werden nach einer fehlerfreien Befehlsübertragung die Bilddaten übertragen. Das Erstellen der Bitmap wird in Kapitel 4.3.2 erklärt. Es werden 160x120 Pixel angefordert und durch das vorliegende RGB565 Format ist der Farbwert von jedem Pixel in 16 Bit kodiert. Es müssen somit 2 Byte pro Pixel übertragen werden.

$$Größe = Höhe * Breite * 2$$
$$Größe = \frac{120}{Pixel} * \frac{160}{Pixel} * 2\frac{Byte}{Pixel}$$
$$Größe = 38400 Byte$$

Formel 2 Berechnung Speichergröße

Damit nicht 38400 Byte (siehe Formel 2) gespeichert werden müssen, wird bei jedem eingelesenen Pixel sofort die Entscheidung getroffen, ob er schwarz (1) oder weiß (0) ist. Diese Entscheidung wird mit dem globalen Schwellwertverfahren[21] realisiert. Dabei wird für jeden der drei Farbbereiche ein Schwellwert festgelegt, ab dem ein Pixel als weiß gilt. Um diese Entscheidung durchführen zu können, müssen die zwei Byte Farbinformation zu einem Wert zusammengefasst werden. Wie im Quellcode zu sehen ist, wurde der Datentyp „uint16_t“ gewählt, da er 16 Bit groß ist und somit genau 2 Byte aufnehmen kann. Die Kamera sendet von jedem Pixel immer erst das untere Byte, also die Bits 0-7, und danach das obere Byte (Bits 8-15). Beide Bytes werden zunächst getrennt gespeichert und dann zusammengeführt, indem das höherwertige Byte auf die oberen acht Bit geschoben wird und das untere Byte dann hinzugefügt wird.

Nun können die einzelnen Farbbereiche maskiert und mit dem Schwellwert verglichen werden. Die Masken sind 0xF800 für rot, 0x7E0 für grün und 0x1f für blau. Der Schwellwert für rot und blau liegt bei 16 (0x10), der für grün bei 32 (0x20). Diese Werte wurden durch manuelles Auswerten des Farbhistogramms (siehe Kapitel 4.2.1) festgelegt. Das Maskieren erfolgt durch eine logische „UND“ Verknüpfung der Farbinformation und einer Maske, die an den Bitstellen der gewünschten Farbe 1 ist und sonst 0. Nach der Maskierung werden die Bits an das untere Ende des Bytes geschoben und mit der Schwelle verglichen.

Als Beispiel wird in Tabelle 6 der Grünwert eines zufälligen RGB Wertes maskiert und verglichen.

[21] globales Schwellwertverfahren: Segmentierung des Bildes aufgrund eines Wertes, der für das gesamte Bild gleich ist.

RGB Wert	00111 001110 00111
Maske grün	00000 111111 00000
Ergebnis UND Verknüpfung	00000 001110 00000
Ergebnis Verschiebung	0000 0000 0000 1110
Schwelle grün	0000 0000 0010 0000
Ergebnis Vergleich	Farbwert < Schwellwert

Tabelle 6 Beispiel Farbmaskierung und Schwellwertvergleich

Nachdem das Maskieren und Vergleichen für alle Farben stattgefunden hat, ist entschieden welche Farbe und somit welchen Wert der Pixel hat. Dieser Wert wird dann in einem 160x120 Bytes großen Array als char Datentyp gespeichert. Auch wenn theoretisch ein Bit pro Pixel ausreichend wäre, ist das Speichern als „char“ wichtig für die weitere Verarbeitung des Bildes. Das Einlesen und Konvertieren ist in der Funktion „uart_gets_cam_dat()“ umgesetzt worden.

Die Kamera sendet die Pixel in Paketen. Jedes Paket wird mit den Zeichen „DAT:“ eingeleitet und enthält eine Zeile (120 Pixel) des Bildes. Die Zeilen werden von rechts nach links in dem Array gespeichert. Die einzelnen Pixel der gesendeten Spalte werden von oben nach unten gespeichert. Das erste Pixel im ersten Paket steht somit in Array Element (0,0 – Spalte, Zeile) und das zweite in Element (0,1). Das nächste Paket wird von Element (1,0) an eingetragen. In dem Quellcode ist zu sehen, dass das Array nicht mit den Koordinaten angesprochen wird. Stattdessen wird der Speicherplatz im SRAM berechnet, an welchem das Pixel abgelegt wird. Da das Array einen festen Speicherbereich hat und jedes Pixel einem Byte entspricht, kann der Speicherplatz für jedes Element mit folgender Formel einfach berechnet werden:

$$Element = Anfangsadresse + Spalte + Zeile * 160$$

Formel 3 Berechnung des Speicherplatzes eines Pixels

Wenn nun Beispielsweise das Element (42,110) belegt werden soll und die Anfangsadresse bei 0x20000000 liegt, dann liegt der Speicherplatz für das Element bei:

$$Element = 0x20000000 + 42 + 110 * 160$$
$$Element = 0x20000000 + 0x2A + 0x6E * 160$$
$$Element = 0x200044EA$$

Formel 4 Beispielberechnung Speicherelement

Die immer wiederkehrende Zeichenkette „DAT:“ wird eingelesen aber nicht gespeichert. Wurden 160 Pakete eingelesen ist das Bild vollständig in dem Array gespeichert. Bevor das Bild eingelesen wird, wird im Terminal der Text „*Bild einlesen*“ ausgegeben. Nachdem das Bild vollständig gespeichert ist, wird „*Bild eingelesen*“ angezeigt.

4.3.2 Aufgenommenes Bild anzeigen

Da die Kamera, wie bereits in Kapitel 4.2.1 erwähnt, keine Möglichkeit bietet das aufgenommene Bild zur Anzeige zu bringen, musste eine indirekte Lösung gefunden werden. Nachdem das Bild vollständig eingelesen und als ein Bit/Pixel Information gespeichert ist, wird aus diesen Informationen ein Bitmap erstellt und dies über die UART-Verbindung an den PC gesendet. Der dafür zuständige Programmteil ist „make_bmp()“. Um aus den Bits ein Bitmap zu generieren, muss eine Header[22] eingefügt werden. Dieser ist 14 Byte groß und enthält folgende Informationen:

Größe	Bedeutung	Wert
2 Byte	Kennung „BM“	0x42, 0x4D
4 Byte	Größe der Datei	0x09, 0x96, 0x00, 0x00
2 Byte	Reserviert	0x00, 0x00
2 Byte	Reserviert	0x00, 0x00
4 Byte	Kennzeichnet den Start der Nutzdaten	0x3E, 0x00, 0x00, 0x00

Tabelle 7 Bitmap Header

[22] Header: Nutzdatenergänzende Zusatzinformationen, welche vor den Nutzdaten eingefügt werden.

Zusätzlich zu dem Header muss noch ein Informationsblock eingefügt werden, welcher die Eigenschaften der Bitmap genau festlegt. Dieser ist 40 Byte groß (+8Byte Farbinformation) und besteht aus folgenden Einträgen:

Größe	Bedeutung	Wert
4 Byte	Länge des Informationsblocks	0x28, 0x00, 0x00, 0x00
4 Byte	Breite des Bildes	0xA0, 0x00, 0x00, 0x00
4 Byte	Höhe des Bildes	0x78, 0x00, 0x00, 0x00
2 Byte	Zahl der Ebenen	0x01, 0x00
2 Byte	Farbbit / Pixel	0x01, 0x00
4 Byte	Kompression	0x00, 0x00, 0x00, 0x00
4 Byte	Größe des Bildes (unkomprimiert)	0x09, 0x60, 0x00, 0x00
4 Byte	Pixel/Meter (horizontal)	0x00, 0x00, 0x00, 0x00
4 Byte	Pixel/Meter (vertikal)	0x00, 0x00, 0x00, 0x00
4 Byte	verwendete Farben	0x00, 0x00, 0x00, 0x00
4 Byte	wichtige Farben	0x00, 0x00, 0x00, 0x00
4 Byte	1. Farbe (rot, grün, blau, reserviert)	0x00, 0x00, 0x00, 0x00
4 Byte	2. Farbe (rot, grün, blau, reserviert)	0xFF, 0xFF, 0xFF, 0x00

Tabelle 8 Bitmap Informationsblock

Die in Tabelle 7 und Tabelle 8 aufgezeigten Werte werden in die ersten 62 Elemente des Bitmaparrays gespeichert, danach folgen die Daten. Diese werden als 1Bit/Pixel gespeichert und somit passen acht Pixel in ein Byte. Die beim Einlesen des Bildes gespeicherten Pixel werden zu Bytes zusammengefasst und ebenfalls im Bitmaparray gespeichert. Dabei ist zu beachten, dass die Reihenfolge der Zeilen in

einer Bitmap Datei in umgekehrter Reihenfolge gespeichert werden. Das so entstandene Array ist 2462 Byte groß (Header+Informationsblock+Daten).

Diese Daten werden über die UART-Verbindung an de PC gesendet. Das dort verwendete Terminalprogramm „HTerm“ kann die übertragenen Daten als Logfile speichern. Dabei werden allerdings nicht nur die Bitmapdaten gespeichert, sondern auch alle Ausgaben, die das Programm vorher schon auf das Terminal geschrieben hat.

Mithilfe des Programms „bitmap_gen.exe“ (Anhang D) werden die Bitmapdaten aus dem Logfile ausgelesen. Dieses muss dafür den Namen „Output“ haben und unter „C:\Documents and Settings\test\Desktop\Output“ zu finden sein.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
B	i	l	d		e	i	n	l	e	s	e	n		\n	\r	B	M
42	69	6C	64	20	65	69	6E	6C	65	73	65	6E	20	0A	0D	42	4D

Abbildung 4.12 Auszug aus Logfile

Abbildung 4.12 zeigt einen Ausschnitt aus dem Logfile. Es ist zu sehen, dass vor den Bitmapdaten noch der Hinweis „*Bild einlesen*“ ausgegeben wurde. Dies ist der Normalfall. Sollte nach diesem Hinweis noch ein weiterer ausgegeben werden, wäre dies eine Fehlermeldung und die Dekodierung wäre abgebrochen worden. Um die Bitmapdaten zu extrahieren werden deshalb die ersten 16 Bytes übersprungen und erst die darauffolgenden 2462 Byte eingelesen.

Diese Daten werden ausgewertet und in einer neuen Datei gespeichert, welche „Code.bmp“ genannt und unter „C:\Documents and Settings\test\Desktop\Code.bmp“ gespeichert wird. Es ist zu beachten, dass die hier angegebenen Pfade nur für den zur Entwicklung genutzten PC gelten und für andere PCs angepasst werden müssen.

Das so erstellte Bild ist die einzige Möglichkeit zu überprüfen, was die Kamera aufgenommen hat.

4.3.3 QR Code auf Bild erkennen

Nachdem die Kommunikation mit der Kamera abgeschlossen ist und das Bild als schwarz/weiß Information vorliegt, beginnt die Suche nach einem auf dem Bild vorhandenen QR Code. Eingeleitet wird dies durch die Terminalanzeige „*Code suchen*“. Auf dem Bild wird nach den Ecksteinen (siehe Kapitel 4.1.2) gesucht, welche das abwechselnd schwarzweiße Muster im Verhältnis 1:1:3:1:1 aufweisen. Dies geschieht in der Funktion „finder_scan()“. Diese ist, wie die nachfolgenden Funktionen auch, in der Datei „identify.c“ zu finden. Die Pixel werden Zeile für Zeile von rechts nach links auf fünf aufeinanderfolgende Farbwechsel überprüft. Die Anzahl der gleichfarbigen Pixel wird gezählt. Aus den ersten und letzten beiden Werten wird der Durchschnittswert berechnet und dann geprüft, ob das Muster richtig ist. Außerdem werden die Stellen, an denen die Farbwechsel auftreten gespeichert. Zum Erkennen und Markieren der zusammenhängenden, schwarzen Flächen wird ein „floodfill“[23] Algorithmus eingesetzt („flood_fill_seed()“), dieser betrachtet die an einen schwarzen Pixel angrenzenden Pixel. Sind diese ebenfalls schwarz, wird dies als eine zusammenhängende Fläche gewertet und alle Pixel dieser Fläche bekommen den gleichen Wert (im Folgenden Flächennummer genannt) zugewiesen („region_code()“). Außerdem wird die Summe der Pixel dieser Fläche bestimmt. Diese Flächennummer ist der Grund, warum ein Pixel nicht als einzelnes Bit gespeichert werden kann, sondern Platz für weitere Information benötigt.

Ein richtiges Muster reicht alleine jedoch nicht aus um einen Eckstein zu erkennen. Es ist wichtig, den äußeren Rahmen und das eingeschlossene Rechteck zu erkennen, da sich diese nicht berühren dürfen und im richtigen Verhältnis zueinander stehen müssen. Dies ist in der Funktion „test_capstone()“ umgesetzt. Zuletzt wird noch das Verhältnis der Flächen von dem eingeschlossenem Rechteck zum Rahmen überprüft. Dazu wird die Summe der Pixel des Rechtecks durch die des Rahmens geteilt. Das Verhältnis beträgt idealer Weise:

$$\frac{Rechteckfläche}{Rahmenfläche} = \frac{9\ Zellen}{24\ Zellen} = \frac{3}{8} = 0{,}375$$

Formel 5 Berechnung Verhältnis Rechteck zu Rahmen

[23] floodfill Algorithmus: ursprünglich Entwickelt um zusammenhängende Pixel mit einer Farbe zu füllen, hier nur zum Erkennen und Markieren von Flächen verwendet

Sind alle oben genannten Kriterien erfüllt, wird der Eckstein in die übergeordnete Struktur „qr“ eingetragen („record_capstone()“). Neben den Flächennummern des äußeren Rahmens und des Rechtecks werden auch die Pixelkoordinaten der Ecken des Rahmens und die des Mittelpunktes des Rechteckes gespeichert („find_region_corners()“).

Die obere, linke Ecke des Rechteckes ist bekannt, da sie bei der Farbwechselüberprüfung gefunden wurde. Diese wird nun als Referenzpunkt gesetzt und die Entfernung zu den Kanten des Rings gemessen. Dies geschieht einmal für die horizontale und einmal für die vertikale Achse, wobei die Beträge der Strecke vom Punkt zu dem Referenzpunkt addiert werden. Der jeweils am weitesten entfernte Punkt wird gespeichert. Aus diesen beiden Punkten ergibt sich die untere, rechte Ecke des Rahmens. Die erste Ecke wird mit der Funktion „find_one_corner()“ gefunden, die nächsten Ecken werden in „find_other_corners()“ gesucht. Um die anderen Ecken zu finden, wird eine Linie zwischen dem Referenzpunkt und der gefundenen Ecke berechnet. Die Punkte, die sich rechts und links am Weitesten von der Linie entfernt befinden sind die Ecken oben rechts und unten links. Sie werden mithilfe des „floodfill“ Algorithmus gefunden. Die letzte noch fehlende Ecke befindet sich auf der Verlängerung der Linie in die andere Richtung.

Nachdem die Koordinaten der Ecken gespeichert sind, wird der Mittelpunkt des Ecksteins gesucht. Dafür wird eine projektive (oder auch perspektivische) Abbildung erzeugt. Diese verknüpft die Pixelkoordinaten (x,y) mit Zellenkoordinaten(u,v) (siehe Abbildung 4.13).

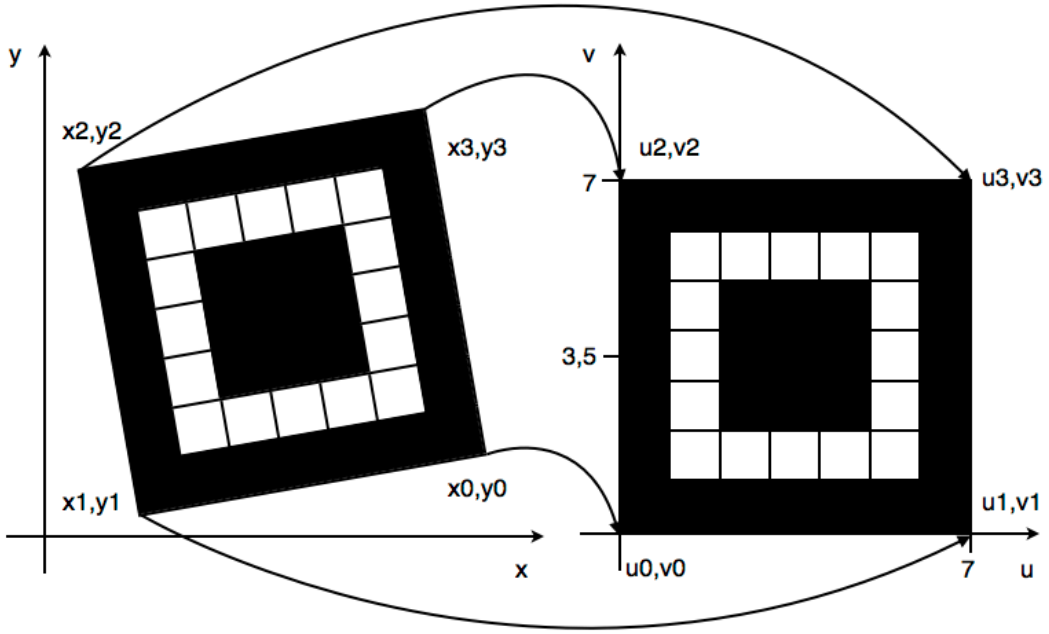


Abbildung 4.13 Projektive Abbildung

Wie in der Abbildung zu sehen ist, verändert sich die Größe des Ecksteins nicht. Die länge der Geraden zwischen 2 Punkten bleibt unverändert. Die vier Ecken des äußeren Rings werden auf die Zellkoordinaten (0,0), (0,7), (7,0) und (7,7) projiziert. Eine eventuell aufgetretene Verdrehung des Musters wird somit ausgeglichen. Die vollständige Erklärung des Algorithmus wird an dieser Stelle nicht geliefert, kann aber beispielsweise in (Burger & Burge 2006, S. 367 ff.) nachvollzogen werden. Mithilfe dieser projektiven Abbildung lassen sich Pixelkoordinaten zu beliebigen Zellen ausrechnen. Dazu werden folgende Formeln benutzt.

$$x = \frac{a_{11} * u * a_{12} * v + a_{13}}{a_{31} * u + a_{32} * v + 1}$$
$$y = \frac{a_{21} * u * a_{22} * v + a_{23}}{a_{31} * u + a_{32} * v + 1}$$

Formel 6 Berechnung der Pixelkoordinaten aus den Zellkoordinaten

Die dazugehörigen Funktionen sind „perspective_setup()“ und „perspective_map()“. Die Mitte des Musters wird somit gefunden wenn in den Formel 6 für u und v der Wert 3,5 angenommen wird. Die acht Abbildungsparameter a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31} und a_{32} können bestimmt werden, da für vier korrespondierenden 2D-Punktpaaren (die Ecken des Rahmens) jeweils ein Punkt $x_i = (x_i, y_i)$ im Ausgangsbild und der zugehörige Punkt $x'_i = (u_i , v_i)$ im Zielbild vorgegeben sind. Daraus lässt sich

ein Gleichungssystem mit acht Gleichungen aufstellen, welches mithilfe eines numerischen Verfahrens gelöst werden muss (vgl. Burger & Burge, 2006, S. 369).

Die Koordinaten der Mitte und die Abbildungsparameter werden ebenfalls in der Struktur „qr“ gespeichert und somit ist ein Eckstein vollständig gespeichert. Dieses Verfahren wird wiederholt, bis das Ende des Bildes erreicht ist. Alle gefundenen Ecksteine werden nach dem oben beschriebenen Verfahren gespeichert. Im Terminal wird „*1 Code gefunden*“ ausgegeben wenn alle drei Ecksteine gefunden wurden. Ansonsten wird die Dekodierung mit der Ausgabe „*kein Code*“ abgebrochen.

4.3.4 Perspektive anpassen und Daten auslesen

Nachdem die drei Ecksteine gefunden und gespeichert wurden, wird in der Funktion „test_grouping()“ geprüft, wo sich diese befinden und daraus die Lage des QR Codes abgeleitet. In der Referenzlage befinden sich die Ecksteine in den oberen beiden sowie in der unteren linken Ecke. Diese Referenzlage muss hergestellt werde, falls der Code nicht in der richtigen Orientierung eingelesen wurde.

Als Erstes wird die Nachbarschaft der Ecksteine überprüft. Der als Erstes gefundene Eckstein dient zunächst als Referenz. Mithilfe der bei der projektiven Abbildung bestimmten Abbildungsparameter der Ecksteine lässt sich der relative Gradient von einem Eckstein zu einem anderen bestimmen. Im Idealfall liegen die Mittelpunkte zweier Ecksteine genau 14 Zellen auseinander. Es wird also überprüft, ob der Referenzeckstein einen Nachbareckstein in vertikaler und horizontaler Ebene hat. Trifft dies nicht zu, wird der nächste Eckstein zur Referenz und es wird erneut getestet. Nur der Eckstein, der sich in der Referenzlage oben links befindet, hat diese Nachbarschaftskonstellation. Die anderen beiden haben nur entweder einen Nachbarn in vertikaler oder in horizontaler Richtung. Zur Vereinfachung werden an dieser Stelle die Bezeichnungen A, B und C für die Ecksteine eingeführt („record_qr_grid()“). Abbildung 4.14 zeigt die Zuordnung.

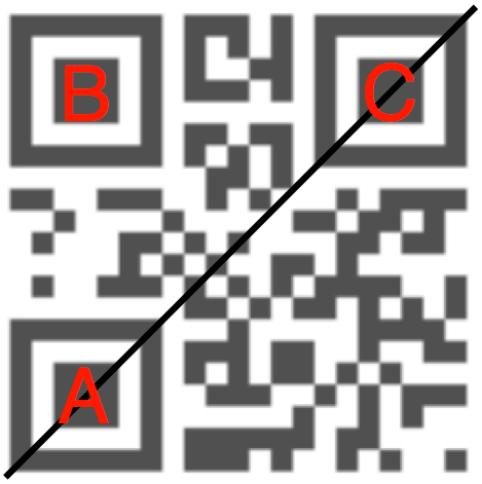


Abbildung 4.14 Zuordnung der Ecksteinbezeichnungen und eingetragene Hypotenuse

Dabei ist zu beachten, dass dies die Bezeichnung in der Referenzlage ist. B ist immer der Eckstein mit den Nachbarn in beiden Richtungen. Je nach Ausrichtung des Codes können A und C vertauscht sein da A immer der vertikal zu B stehende ist und C demnach immer horizontal zu B steht.

Als nächstes wird überprüft ob die Ecksteine im Uhrzeigersinn angeordnet sind. Dazu wird zunächst eine Hypotenuse von A nach C berechnet (siehe Abbildung 4.14). Der Eckstein B sollte sich links von der Linie befinden. Ist dies nicht der Fall werden die Bezeichnungen A und C vertauscht. Unabhängig von der Ausgangsausrichtung des Codes könne die Ecksteine nun in der Referenzlage gespeichert werden.

Durch das Vertauschen der Ecksteine stimmen die vorher bestimmten Ecken nicht mehr zu der Ausrichtung des jeweiligen Ecksteins. Um eine definierte Lage der Ecken zu bekommen, werden die Ecksteine neu ausgerichtet. Beim Finden der Ecksteine wurden die Ecken in der in Abbildung 4.15 aufgezeigten Reihenfolge gespeichert.

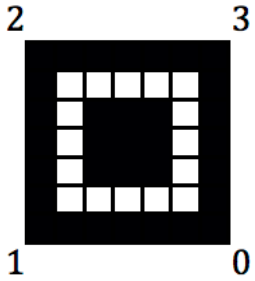


Abbildung 4.15 Reihenfolge der Ecken der gefundenen Ecksteine

Diese werden jetzt so gedreht, dass die ursprünglich untere rechte Ecke (Ecke 0) zu der in der Referenzlage oben links liegenden wird (Ecke 2). Dafür wird mithilfe der Referenzlinie zwischen A und C bestimmt, welche Ecke am weitesten links liegt. Daraus kann ermittelt werden wie der Eckstein gedreht werden muss.

Beispiel:

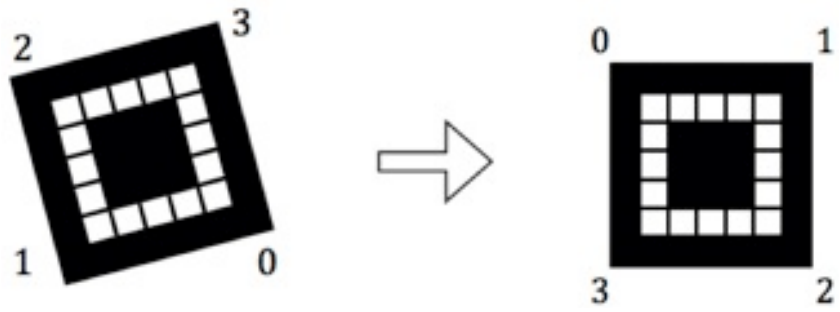


Abbildung 4.16 Eckstein von Ausgangslage in gewünschte Lage bringen

Der in Abbildung 4.16 links gezeigte Eckstein soll in die in rechts gezeigte Lage gebracht werden. Es ist anzunehmen, dass die Ecke zwei in der Ausgangslage die am weitesten links von der Referenzlinie liegende ist. Sie wird x genannt. Die Sortierung der Ecken erfolgt nach folgendem Prinzip („rotate_capstone()“):

$$neueEcke[i] = (Ecke[i] + x)\%4$$
$$neueEcke[0] = (0+2)\%4 = 2$$
$$neueEcke[1] = (1+2)\%4 = 3$$
$$neueEcke[2] = (2+2)\%4 = 0$$
$$neueEcke[3] = (3+2)\%4 = 1$$

Formel 7 Ecken drehen

Es ist zu erkennen, dass die Zuordnung durch eine einfache modulo Operation erreicht wird. Mit den gedrehten Ecken wird wieder eine projektive Abbildung vorgenommen um die neuen Abbildungsparameter der Ecken zu bestimmen. Die neu ausgerichteten Ecksteine werden in die Struktur QR übernommen und ersetzen damit die ursprüngliche Reihenfolge und Orientierung. Der Code liegt jetzt in der Referenzlage.

Anhand der Referenzlage können die „timing patterns“ geprüft werden („measure_timing_pattern()“), da die Lage dieser vorgegeben ist. Die Größe dieser

Muster legt die Größe des Referenzgitters fest. Der Startpunkt beider „timing pattern“ liegt in Ecke 0 des Ecksteins B. Um ein möglichst gutes Ergebnis zu bekommen, wird die Pixelkoordinate in der Mitte der Zelle (7/7) des Ecksteins B berechnet. Dafür wird die projektive Abbildung für den Punkt (6,5/6,5) bestimmt. In horizontaler Richtung trifft das Muster auf die Ecke 1 des Ecksteins C. Auch hierfür wird die Pixelkoordinate der Mitte der Zelle bestimmt. Diese liegt bei (0,5/6,5). In vertikaler Richtung liegt der Endpunkt in Ecke 3 des Ecksteins A an der Position (6,5/0,5).

Mithilfe eines modifizierten Bresenham Algorithmus[24] wird die Linie zwischen dem Startpunkt und einem der Endpunkte abgetastet und die schwarz-weiß Wechsel ausgewertet. Zunächst wird der Abstand vom Start- zum Endpunkt in x- und y-Richtung bestimmt. Die Richtung, welche dem Betrag nach den größeren Abstand aufweist, wird zur Längslaufrichtung die andere zur diagonalen Laufrichtung. Die Linie wird schrittweise in Längsrichtung abgelaufen, wobei das Vorzeichen des Inkrements abhängig von dem Vorzeichen des Abstandes ist. Um die Schritte in diagonale Richtung fest zu legen, wird bei jedem Längsschritt eine Variable um den Abstand der diagonalen Richtung erhöht. Ist dieser Wert größer als der Abstand in Längsrichtung wird ein Schritt in die diagonale Richtung gemacht. Die Richtung wird ebenfalls durch das Vorzeichen des Abstandes festgelegt. Nach dem Schritt wird der Abstand der Längsrichtung von der Variablen abgezogen. Ein Beispiel hierfür ist in Abbildung 4.17 zu sehen.

[24] Bresenham Algorithmus: ursprünglich zum Zeichnen von Geraden erfunden, benannt nach dem Entwickler Jack Bresenham.

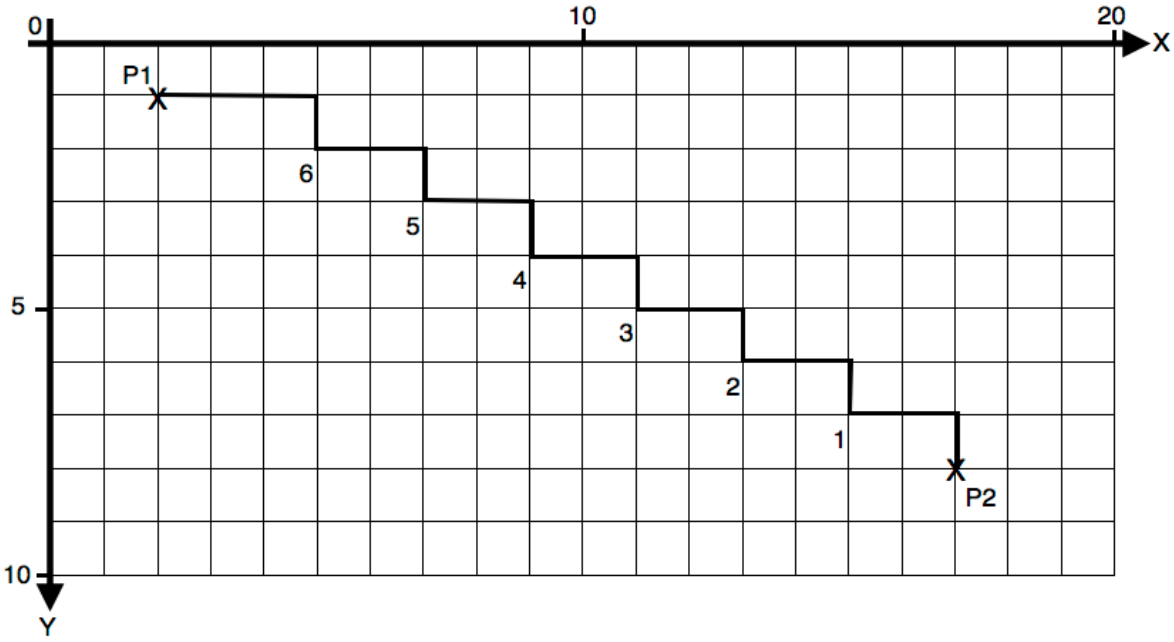


Abbildung 4.17 Beispiel Bresenham Algorithmus

In dem angegeben Beispiel sind der Startpunkt (P1, 2/1) und der Zielpunkt (P2, 17/8) 15 Schritte in x-Richtung und sieben Schritte in y-Richtung voneinander entfernt. Damit liegt die x-Richtung als Längs- und die y- als diagonale Richtung fest. Die Inkremente sind in beiden Richtungen positiv. Pro Schritt in x Richtung wird eine Variable a um sieben erhöht. Der erste Schritt in y Richtung wird nach drei Schritten in x Richtung eingefügt, da a=3*7=21 ist und somit größer als 15. Die in der Abbildung 4.17 unterhalb der Linie eingetragenen Werte sind die Ergebnisse aus der Berechnung a=a-15, die nach jedem Schritt in y-Richtung ausgeführt wird.

Die Farbe des Pixels wird bei jedem Schritt überprüft und so festgestellt, wie viele schwarz-weiß Wechsel vorkommen. Gezählt werden nur die Wechsel von Weiß zu Schwarz. Diese Überprüfung wird bei beiden „timing patterns“ durchgeführt. Unterscheidet sich die Anzahl der gefundenen Farbwechsel, wird die größere weiter verwendet. Anhand dieser Angabe kann mithilfe der Formel 8 die Größe (in Zellen) und somit auch die Variante des QR Codes berechnet werden (x = Anzahl der gezählten weiß-schwarz Wechsel).

$$Größe = x * 2 + 13$$

$$Variante = \frac{(Größe - 15)}{4}$$

Formel 8 Berechnung Größe und Variante

Die in dieser Arbeit eingesetzte Variante 1 hat immer eine Größe von 21 Zellen. Die Anzahl der gefundenen Wechsel sollte somit vier betragen. Die Anzahl der weiß-schwarz Wechsel, die Größe und die Variante werden ebenfalls in die Struktur QR eingetragen.

Damit das Referenzgitter möglichst genau positioniert werden kann, wird noch der Punkt bestimmt, an dem ein „alignment Pattern“ auftreten würde („line_intersect()“). Verlängert man die linke Seite von Eckstein C und die obere Kante von Eckstein A, befindet sich dieser Punkt in dem Schnittpunkt dieser Linien.

Nach dieser Berechnung können die Abbildungsparameter der projektiven Abbildung des Referenzgitters bestimmt werden („setup_qr_perspektive()“), da vier Punkte eindeutig bestimmt sind (siehe rote Markierungen in Abbildung 4.18).

Abbildung 4.18 Punkte zur Bestimmung des Referenzgitters

Bei Codes der Variante 1 könnte auch die untere rechte Ecke bestimmt und das Referenzgitter anhand der Ecken des Codes berechnet werden. Ein solches Vorgehen wird bei größeren Varianten durch den größeren Abstand ungenau. Da eine mögliche Erweiterung des Programms nicht ausgeschlossen ist, wird auch für Variante 1 die oben beschriebene Möglichkeit eingesetzt.

Mithilfe der so bestimmten Abbildungsparameter des Referenzgitters können die Ecken des Codes berechnet werden. Dafür werden die Pixelkoordinaten der Zellen (0/0), (21/0), (0/21) und (21/21) berechnet und gespeichert („quirc_extract()“).

Der QR Code liegt jetzt in der Referenzlage vor. Die Eckstein sind bestimmt und anhand der „timing Patterns“ wurden die Größe und Variante bestimmt. Mithilfe des

Referenzgitters wurden die Ecken des Codes berechnet. Alle diese Informationen wurden in der Struktur QR gespeichert.

Im nächsten Schritt können nun die Zellen anhand des Referenzgitters ausgewertet werden. Die Zellen werden Zeile für Zeile von links nach rechts überprüft. Eine schwarze Zelle wird als 1, eine weiße als 0 gespeichert. Um an die Farbinformation einer bestimmten Zelle (u/v) zu gelangen, sind folgende Berechnungen nötig:

$$p = v * Größe + u$$
$$Zelle = (ArrayElement[p >> 3] >> (p \& 7)) \& 1$$

Formel 9 Bestimmung der Information von Zelle u/v

Die Farbinformation wird in einem „char“ Array gespeichert, wobei jede Zelle einem Bit entspricht. In jedem Element diese Arrays werden acht Zellen gespeichert, wobei die achte Zelle das MSB darstellt. Das erste Arrayelement ist beispielsweise immer mit 0b0111 1111 belegt, da als erstes die obere Kante des Ecksteins mit sieben schwarzen Zellen eingelesen wird und danach immer eine weiße Zelle folgt. Um mögliche Fehler an den Kanten der Zellen zu umgehen, wird immer das Pixel in der Mitte einer Zelle überprüft. Für die erste Zelle wird beispielsweise die Pixelkoordinate der Zelle (0,5/0,5) bestimmt und überprüft. Da die Zellenanzahl nie ganzzahlig durch 8 teilbar ist, wird im letzen Arrayelement nur das letzte Bit verwendet; die restlichen Bits werden mit 0 aufgefüllt.

4.3.5 Daten dekodieren und ausgeben

Aus den gespeicherten Informationen der Zellen sollen nun die Daten dekodiert werden. Die dazu notwendigen Funktionen befinden sich in der Datei „decode.c“ Dazu sind folgende Schritte notwendig (siehe Abbildung 4.19).

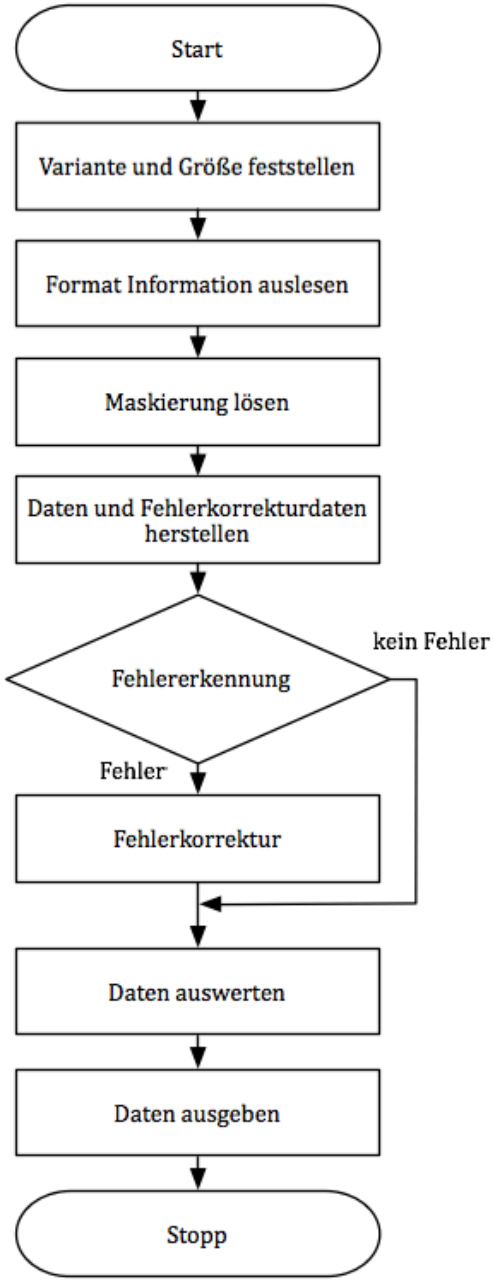


Abbildung 4.19 Flussdiagramm Daten dekodieren

Wie im Flussdiagramm zu sehen ist, wird zuerst die Variante und Größe festgestellt („quirc_decode()“). Diese Informationen wurden, wie in Kapitel 4.3.4 bereits beschrieben, ausgelesen und werden nun überprüft. Die Größe des Codes ist zulässig, wenn bei der Berechnung $(Größe - 17)\%4$ kein Rest bleibt. Die Variante darf nicht kleiner eins und nicht größer 40 sein. Wird eines dieser Kriterien nicht erfüllt, wird das Dekodieren abgebrochen und der Fehlertext „*Invalid grid size*“ bzw. „*Invalid version*“ ausgegeben. Sind alle Kriterien erfüllt, werden als nächstes die

Formatinformationen ausgewertet („read_format()“ siehe Anhang E). Die Auswertung der beiden Positionen erfolgt unabhängig. Zuerst wird die Position bei Eckstein B überprüft. Tritt dabei ein Fehler auf, wird die zweite Position getestet. Wird beide Male ein Fehler erzeugt, wird die Dekodierung mit der Fehlermeldung „*Format data ECC failure*“ beendet.

Wie bereits in Kapitel 4.1.2 erwähnt, wird die Formatinformation mithilfe einer Tabelle umgesetzt. Dies vereinfacht die Dekodierung, da der BCH(15,5) Code nicht entschlüsselt werden muss. Es gibt 32 gültige Bitsequenzen, welche in einer Tabelle im Speicher des Mikrocontrollers abgelegt sind. Die eingelesenen Bits der Formatinformation werden erst demaskiert, indem sie mit 0x5412 XOR verknüpft werden. Das Ergebnis wird Bit für Bit mit den 32 Elementen verglichen. Die Hamming Distanz des Fehlerkorrekturcodes ist sieben, somit kann die eingelesene Information mit maximal drei falschen Bits immer noch korrekt zugeordnet werden. Der passende Tabelleneintrag wird ausgewertet, wobei die ersten beiden Bits das Fehlerkorrektur-level angeben und die nächsten drei Bits die Art der Maskierung festlegen. Diese Informationen werden für die weitere Dekodierung gespeichert.

Als nächstes werden die zuvor gespeicherten Zelleninformationen in die entsprechenden Daten- und Fehlerkorrektursymbole (siehe Abbildung 4.4) sortiert („read_data()“). Gestartet wird mit der Zelle in der unteren linken Ecke (21/21). Da nur die Zellen benötigt werden, die Daten beinhalten, wird bei jeder Zelle überprüft, ob sie zu einem der Ecksteine, der Formatinformation oder den „timing pattern“ gehört. Ist dies nicht der Fall, kann die Information der Zelle mit Hilfe der Formel 9 ausgelesen werden. Danach wird überprüft, ob die Zelle maskiert ist. Dafür wird die Art der Maskierung aus der Formatinformation ausgewertet und die entsprechenden Gleichung (siehe Abbildung 4.5) für die entsprechende Zelle gelöst. Ist das Ergebnis Null, wurde die Zelle maskiert und die Information muss invertiert werden. Die so entstandenen Bits werden zu Bytes zusammengefasst und entsprechend der Symbole gespeichert.

Für die Fehlerkorrektur werden die Symbole zu einem Block zusammengefasst, in welchen zuerst die Datensymbole und danach die Fehlerkorrektursymbole eingetragen werden. Beim Erstellen eines QR Codes wird ein Reed-Solomon

Algorithmus[25] benutzt, um eine Fehlererkennung und Behebung zu ermöglichen. Beim Dekodieren werden die Daten wie oben beschrieben eingelesen und gespeichert und dann mithilfe eines Berlekamp-Massey Algorithmus[26] überprüft und gegebenenfalls geändert („codestream_ecc()“). Kann die Fehlerkorrektur nicht ordnungsgemäß durchgeführt werden, wird die Dekodierung mit der Ausgabe „*ECC failure*“ beendet.

Nach der Fehlerkorrektur liegen die korrigierten Daten als Bytes in einem Array und können ausgewertet werden („decode_payload()“). Wie bereits erwähnt, gibt es vier unterstütze Zeichensätze. Die Information, zu welchem Zeichensatz die vorliegenden Daten gehören, liefern die ersten vier Bit im ersten Byte. Dabei gilt folgende Zuordnung (siehe Tabelle 9):

Numerisch	0b0001
Alphanumerisch	0b0010
Byte	0b0100
Kanji	0b1000

Tabelle 9 Kennzeichnung der Zeichensätze

Nach diesen vier Bit folgt unabhängig des Zeichensatzes die Angabe, wie viele Zeichen in dem Code kodiert sind. Diese Angabe wird überprüft und wenn sie größer als 26 (max. zulässige Zeichenzahl) ist, wird die Dekodierung mit dem Fehler „*Data overflow*“ beendet. Außerdem wird überprüft, ob die Anzahl der Bits inklusive Zeichensatzinformation und Zeichenanzahl Angabe kleiner als 152 (8 Bit *19 Datensymbole = 152) ist. Wird dieser Wert überschritten, wird „*Data underflow*“ ausgegeben und die Dekodierung ebenfalls beendet.

[25] Reed-Solomon Algorithmus: Kodierungsverfahren; Bei Daten, die so kodiert wurden, können Fehlererkennungen und -korrekturen durchgeführt werden, benannt nach den Entwicklern I. S. Reed und G. Solomon

[26] Berlenkamp-Massey Algorithmus: Decodierung des Reed-Solomon Algorithmus, benannt nach den Entwicklern E. Berlenkamp und J. Massey

Je nach Zeichensatz erfolg die Auswertung der Information auf verschiedene Art. Im Folgenden wird die Auswertung für jeden Zeichensatz erklärt.

Numerischer Zeichensatz („decode_numeric()“):

Bei der numerischen Kodierung werden immer drei Zeichen zusammengefasst und die binäre Darstellung der daraus entstandenen Gruppe mit zehn Bit in den Code eingetragen. Ist die Anzahl der Zeichen nicht ganzzahlig durch drei teilbar, werden die letzen beiden Zeichen zusammengenommen und als Sieben-Bit-Zahl dargestellt. Bleibt nur ein Zeichen über wird dieses mit vier Bit eingetragen.

Die Zeichenkette 0123 ergibt somit die binäre Darstellung:

- 012 → 00 0000 1100
- 3 → 0011
- zusammen: 00 0000 1100 0011

Die Angabe, wie viele Zeichen im Code enthalten sind, wird mit zehn Bit eingetragen.

Zum Dekodieren werden immer zehn Bit ausgelesen. Die daraus entstehende Zahl wird modulo 10 gerechnet, um an die letzte Stelle der Zahl zu gelangen. Diese wird in das Array Element x+2 gespeichert. Durch Teilen der Zahl durch zehn wird die letzte Stelle abgetrennt. Durch erneutes modulo 10 Rechnen wird wieder die letzte Ziffer der Zahl berechnet. Diese wird in das Arrayelement x+1 gespeichert. Dieser Vorgang wird nochmals wiederholt, wobei die zuletzt gefundene Stelle in das Element x gespeichert wird. Ist die letzte Zeichengruppe nicht mehr drei Zeichen groß, wird durch Vergleichen der Anzahl der bereits verarbeiteten Zeichen mit der Gesamtanzahl der Zeichen sichergestellt, dass nur die zu den Daten gehörenden Zeichen verarbeitet werden.

Alphanumerischer Zeichensatz („decode_alpha()“):

Diese Kodierung entsteht mithilfe einer Tabelle, in welcher alle Zeichen, die vorkommen können, jeweils einem Dezimalwert zwischen Null und 44 zugewiesen werden. Die binäre Darstellung wird erreicht, indem jeweils zwei der Dezimalzahlen zusammengerechnet werden. Dabei wird die erste mit 45 multipliziert und die zweite

zu diesem Ergebnis addiert. Das Ergebnis dieser Berechnung wird als Elf-Bit-Zahl binär in den Code eingetragen. Ist die Anzahl der Zeichen nicht ganzzahlig durch zwei teilbar, wird das letzte Zeichen als Sechs-Bit-Binärdarstellung eingetragen.

Die Anzahl der im Code vorhandenen Zeichen wird als Neun-Bit-Zahl dargestellt.

Zum Dekodieren werden immer elf Bit eingelesen. Die so entstandene Dezimalzahl wird modulo 45 gerechnet, um die hintere Zahl der Zweiergruppe zu berechnen. Das Ergebnis wird in dem Arrayelement x+1 gespeichert. Durch Teilung mit 45 und erneutes modulo 45 Rechnen wird die erste Zahl der Gruppe bestimmt, welche in dem Element x gespeichert wird. Die so ermittelten Werte werden dann anhand der Tabelle dem auszugebenden Zeichen zugeordnet.

Byteweise („decode_byte()“):

Bei der byteweisen Kodierung repräsentiert ein Byte immer auch ein Zeichen.

Nach den vier Bit Information zu dem Zeichensatz folgen acht Bit, die angeben, wie viele Zeichen in dem Code gespeichert sind. Im Anschluss daran folgen direkt die Zeichen. Dabei ist zu beachten, dass nicht jedes Datensymbol ein ganzes Zeichen enthält. Durch die vier Bit Zeichensatzinformation befinden sich die oberen vier Bit in Datensymbol x und die unteren vier Bit in Datensymbol x+1. Die Zeichen werden wieder zu einem Byte zusammengefügt und in einem Array gespeichert.

Kanji Zeichensatz:

Die Codierung von Kanji Zeichen erfolgt mithilfe der Shift-JIS (Japanese Industrial Standard) Zeichensatztabelle. Diese ist ein Bestandteil der ASCII Tabelle. Es gibt zwei Bereiche, in denen die Schriftzeichen liegen: der eine von 0x8140-0x9FFC, der andere von 0xE040-0xEBBF. Jedes Schriftzeichen wird durch zwei Byte dargestellt. Bei der Kodierung wird je nach Bereich 0x8140 oder 0xE040 von der Zweibytedarstellung des Zeichens subtrahiert. Danach wird das höherwertige Byte mit 0xC0 multipliziert und das untere Byte zu dem Ergebnis addiert. Das Ergebnis dieser Berechnung wird als 13-Bit-Binärdarstellung in den Code eingetragen.

Zum Dekodieren werden immer 13 Bit ausgelesen und die oben beschriebenen Berechnungen rückgängig gemacht. Die zwei Byte eines Zeichens werden in zwei Arrayelementen gespeichert.

Nach dem Auswerten der Zeichen erfolgt die Ausgabe. Dies geschieht wie bei den Fehlerausgaben auch im Terminal. Beispielhaft wird hier (Abbildung 4.20) die Ausgabe für den in Abbildung 4.1 gezeigten QR Code abgebildet.

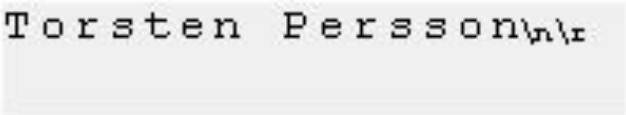


Abbildung 4.20 Ausgabe im Terminalprogramm

Die Dekodierung des eingelesen QR Codes ist nach der Ausgabe beendet. Der Mikrocontroller wartet auf einen erneuten Tastendruck

4.3.6 Belegung des Speicherplatzes

Der Speicherplatz auf dem Mikrocontroller ist auf 96kB SRAM beschränkt. Diesen Platz müssen sich Variablen und der Stack teilen. Bei einer Auflösung von 160*120 Pixel belegt das Bild 19200 Byte. Die Größe des Stacks wurde auf 10 kByte erhöht (Beschreibug dazu Anhang F), da der „floodfill“ Algorithmus viele Daten darauf ablegen muss und die Rücksprungadressen und Parameter der Funktionsaufrufe gespeichert werden müssen. Im Anhang G ist die vom Linker ausgegeben Datei enthalten, welche die Speicherplatzbelegung zeigt. In der vorliegenden Version des Quellcodes sind 31452 Byte des SRAMs belegt.

4.4 Festlegung der Rahmenbedingungen

Es gibt für die Größe eines QR Code keine Beschränkungen. Somit ist diese hauptsächlich von der gewählten Kamera abhängig. Für die hier eingesetzte CMUcam4 wurde die optimale Größe des Codes im Verhältnis zu dem Abstand zur Kamera experimentell bestimmt. Dabei hat sich herausgestellt, dass eine Codegröße

von 5*5 cm ideal ist. Der Abstand zur Kamera sollte bei dieser Größe zwischen 10cm und 12cm liegen. Die Anzahl der Pixel pro Zelle liegt dann bei circa 3*3 (Abbildung 4.21).

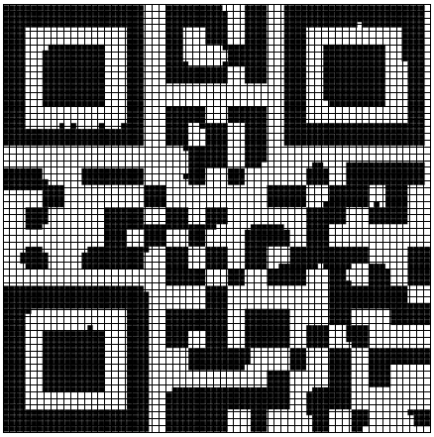


Abbildung 4.21 QR Code aus einer von der Kamera erstellten Bitmapdatei

Abbildung 4.21 zeigt den vergrößerten Ausschnitt aus einer von dem Board erstellten Bitmapdatei. Die Bitmap in Abbildung 4.22 ist in Originalgröße angegeben. Die Kantenlänge des aufgenommenen QR Codes beträgt 5cm, der Abstand zur Kamera 12cm.

Abbildung 4.22 Bitmap in Originalgröße

Bei optimaler Beleuchtung und rechtwinkliger Position kann der Abstand bis zu 20cm betragen. Minimal sind 8cm Abstand möglich, jedoch funktioniert das Erkennen des Bildes dann nicht mehr zuverlässig.

Es können auch QR Codes mit anderen Kantenlängen eingelesen werden. Je größer diese wird, desto größer muss der Abstand zur Kamera gewählt werden. Dadurch steigt der Einfluss der Beleuchtung und es treten mehr Fehler auf.

Bei kleineren Kantenlängen muss der Abstand zu der Kamera verringert werden, was ebenfalls zu mehr Fehlern führt.

Eine Verdrehung des Codes kann bis zu einem gewissen Level auch erkannt werden. Drehungen um 90 oder 180 Grad stellen kein Problem dar.

Drehungen zwischen 30 und 45 Grad dagegen führen dazu, dass der Code nicht mehr erkannt wird. Die in Abbildung 4.23 und Abbildung 4.24 gezeigten Codes lassen sich ohne Probleme dekodieren. Die Bitmaps sind in Originalgröße angezeigt.

Abbildung 4.23 Bitmap eines um 180 Grad verdrehten Codes

Abbildung 4.24 Bitmap eines um circa 17 Grad verdrehtes Bild

5 Zusammenfassung und Ausblick

Das Ziel dieser Bachelorthesis war es, eine optische Auswertung von ein- und zweidimensionale Codes mithilfe des ARM Cortex M3 Mikrocontrollers zu realisieren. Es wurde zunächst ein genereller Überblick über die heutzutage verwendeten Codes gegeben und die unterschiedlichen Merkmale anhand von Beispielen verdeutlicht. Im weiteren Verlauf wurden der Aufbau, die besonderen Merkmale, die zur Dekodierung dieser Codes notwendigen Schritte sowie die Umsetzung dieser als Mikrocontrollerprogramm beschrieben. Außerdem wurde ein Überblick über die gewählte Hardware gegeben.

Die Dekodierung des eindimensionalen Code 39 wurde nach den Vorgaben eines Laborversuchs erstellt, der für einen anderen Mikrocontroller entworfen wurde. Es hat sich gezeigt, dass sich die Aufgabe ohne Einschränkungen auf das Stellaris LM3S9B92 Board übertragen lässt. Die Hardware zum Einlesen der Codes kann ohne Änderungen übernommen werden und das Erkennen und Auswerten der Balken funktioniert zuverlässig. Die Durchführung des Laborversuches mit dem Stellaris LM3S9B92 Board wird somit als uneingeschränkt möglich beurteilt.

Bei der Dekodierung von QR Codes müssen dagegen einige Einschränkungen beachtet werden. Es können nur QR Codes des „Model 2“ in der Variante 1 dekodiert werden. Die Gründe hierfür liegen sowohl bei der verwendeten Kamera, als auch bei den Eigenschaften des Stellaris LM3S9B92 Boards.

Um weitere Varianten zu unterstützen, muss die Auflösung der Kamera höher gewählt werden, damit die Codes genauso gut erkannt werden. Daraus ergeben sich mehrere Probleme. Zum Einen vervierfacht sich bei einer Verdopplung der Auflösung der Speicherplatzbedarf für das Array, in dem die Bilddaten gespeichert werden. Das kann ein Überschreiten des zur Verfügung stehenden Speicherplatzes zur Folge haben.

Zum Anderen vervierfacht sich bei einer Verdopplung der Pixel ebenfalls die Zeit, welche der Mikrocontroller benötigt um ein QR Code zu erkennen.

Außerdem benötigt die Kamera doppelt so lange um ein Bild mit der doppelten Auflösung einzulesen.

Eine weitere Einschränkung ist, dass das Bild, auf dem der einzulesende QR Code aufgedruckt ist, für die gesamte Zeit, in der das Bild eingelesen wird (circa 15 Sekunden), ruhig und in konstantem Abstand vor der Kamera sein muss.

Ein Teil dieser Probleme kann durch eine andere Kamera behoben werden, wobei zu beachten ist, dass der USB OTG Treiber des Stellaris LM3S9B92 Board keine Videogeräte unterstützt.

In einer Erweiterung dieser Aufgabe könnte eine LCD Anzeige an das Board angeschlossen werden, um das aufgenommene Bild und den dekodierten Inhalt anzuzeigen. Dies würde den hier umgesetzten Weg über den PC ersetzen.

Abschließend lässt sich festhalten, dass mit den oben beschriebenen Einschränkung das Ziel dieser Arbeit erfüllt wurde.

6 Literaturverzeichnis

Agyeman, K. W., & Rowe, A. (2012). *CMUcam4 Bord Layout and Ports* . Pittsburgh, Pennsylvania, Vereinigte Staaten: Carnegie Mellon University.

Beer, D. (2012). *dlbeer.co.nz*. Abgerufen am 29. 12 2012 von http://www.dlbeer.co.nz/oss/quirc.html

Biermann, K. (21. 09 2011). *Zeit Online*. Abgerufen am 03. 12 2012 von http://www.zeit.de/digital/datenschutz/2011-09/qr-code-hack/komplettansicht

Burger, W., & Burge, M. J. (2006). *Digitale Bildverarbeitung - Eine Einführung mit Java und ImageJ.* Berlin Heidelberg New York: Springer-Verlag.

Hammer, T. (24. 11 2008). *Der-Hammer.info*. Abgerufen am 01. 01 2013 von http://www.der-hammer.info/terminal/index.htm.

Lenk, B. (2002). *2D-Codes Handbuch der automatischen Identifikation* (Bd. 2). Kirchheim unter Teck: Monika Lenk Fachbuchverlag.

o.V. (20. 06 2012). *How to install the terminal plugin in CCSv5*. Abgerufen am 01. 01 2013 von http://processors.wiki.ti.com/index.php/How_to_install_the_terminal_plugin_in_CCSv5

Schneider, J. (12 2004). Ansteuerung eines Barcode-Lesestifts. Hamburg, Deutschland: HAW Hamburg, Labor für Informationstechnik.

Texas Instruments (5. 07 2011). *Stellaris LM3S9B92 Evaluation Kit User's Manual* . Austin, Texas, Vereinigte Staaten: Texas Instruments Incorporated.

Texas Instruments (20. 01 2012). *Stellaris LM3S9B92 Mikrocontroller Data Sheet* . Austin, Texas, Vereinigte Staaten: Texas Instruments Incorporated.

Woodland, N. J., & Silver, B. (1952). *Patentnr. 2612994.* Vereinigte Staaten.

A Quellcode: Code39

```
1  /*-------------------------------------------------------------------------------
2   *  Barcode Scanner
3   *
4   *  Persson       -2012-
5   *-------------------------------------------------------------------------------
6   */
7  #include "stdlib.h"
8  #include "string.h"
9  #include "inc/lm3s9b92.h"
10 #include "inc/hw_memmap.h"
11 #include "inc/hw_types.h"
12 #include "driverlib/debug.h"
13 #include "driverlib/gpio.h"
14 #include "driverlib/pin_map_LM3S9B92.h"
15 #include "driverlib/rom.h"
16 #include "driverlib/sysctl.h"
17 #include "utils/uartstdio.h"
18 #include "driverlib/uart.h"
19 #include "driverlib/systick.h"
20 #include "driverlib/interrupt.h"
21 #include "driverlib/timer.h"
22
23 #ifdef DEBUG//
24 //*****************************************************************************
25
26 //*****************************************************************************
27 //
28 // The error routine that is called if the driver library encounters an error.
29 //
30 void
31 __error__(char *pcFilename, unsigned long ulLine)
32 {
33 }
```

```
34  #endif
35
36  int anz_timer_value = 0, time = 0; // Anzahlt der Balken, Balkenbreite
37  int char_cnt = 0; // Anzahl der Zeichen
38  int timer_value[680]; // Speichert die Breite der Balken
39  short code[75]; // Speichert die Dezimalwerte der Zeichen
40  int decode = 0; // Freigabe der decode_to_char() Funktion
41  int full = 0; // =1 wenn mehr als 675 Flanken detektiert wurden
42
43  //Interrupt handler Pa7
44  //misst die Zeit zwischen 2 Interrupts und speichert diese in einem Array
45  void balkenbreite(void) {
46  //          time = TimerValueGet(TIMER1_BASE, TIMER_A);
47
48  //          SysCtlPeripheralReset(SYSCTL_PERIPH_TIMER1);
49  //          TimerConfigure(TIMER1_BASE, TIMER_CFG_A_PERIODIC_UP);
50  //          TimerEnable(TIMER1_BASE, TIMER_A);
51
52              time = TimerValueGet(TIMER2_BASE, TIMER_A);
53
54              SysCtlPeripheralReset(SYSCTL_PERIPH_TIMER2);
55              TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC_UP);
56              TimerEnable(TIMER2_BASE, TIMER_BOTH);
57
58              if (anz_timer_value <= 679) {//675 = 75*9
59                      timer_value[anz_timer_value] = time;
60                      anz_timer_value++;
61              } else
62                      full = 1;
63
64              GPIOPinIntClear(GPIO_PORTA_BASE, GPIO_PIN_7);
65  }
66
67  //wartet 0,5s
68  void wait(void) {
69              volatile int tmp;
70              for (tmp = 0; tmp < 800000; tmp++);}
71  //Timer settings
```

```
72  void timer_ini(void) {
73          //Timer 1 freischalten
74          SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);
75          //Timer 1 16 Bit, periodisch hochzählen
76          TimerConfigure(TIMER1_BASE, TIMER_CFG_A_PERIODIC_UP);
77          //Timer 2 freischalten
78          SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);
79          //Timer 2 32 Bit, periodisch hochzählen
80          TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC_UP);
81  }
82
83  //Interrupt settings
84  void interrupt_ini(void) {
85          // Pin 7 an Port A als Eingang
86          GPIOPinTypeGPIOInput(GPIO_PORTA_BASE, GPIO_PIN_7);
87          // Pin 7 / Port A als Interruptquelle auf beide Flanken
88          GPIOIntTypeSet(GPIO_PORTA_BASE, GPIO_PIN_7, GPIO_BOTH_EDGES);
89          // Pin7/Port A Intterrupt erlauben
90          GPIOPinIntEnable(GPIO_PORTA_BASE, GPIO_PIN_7);
91          // ISR setzen
92          GPIOPortIntRegister(GPIO_PORTA_BASE, balkenbreite);
93          IntEnable(0); // PortA Interrups erlauben
94          IntMasterEnable(); // global Interrups erlauben
95  }
96
97  void uart_ini() {
98          // UART0 einschalten
99          SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
100         // Port A freigeben
101         SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
102         GPIOPinConfigure(GPIO_PA0_U0RX); // Pin 0 als Receive
103         GPIOPinConfigure(GPIO_PA1_U0TX); // Pin 1 als Transmit
104         // Port A Pin 0 und 1 für UART0
105         GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
106         // Baudrate: 115200
107         UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
108                         (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
109 UART_CONFIG_PAR_NONE));
```

```
110             UARTStdioInit(0);
111     }
112
113     //Port Einstellungen für LED D2
114     void port_ini(void) {
115             // Port  Clock Gating Control
116             SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOD;
117             wait();
118             // Set direction
119             GPIO_PORTD_DIR_R = 0xFF;
120             // Digital enable
121             GPIO_PORTD_DEN_R = 0xFF;
122
123     }
124
125     //entscheidet ob breiter (1) oder schmaler(0) Balken und speichert
126     // jede Ziffer in einem Arrayelement
127     void decode_to_array(void) {
128             int i = 0, timer_value_position = 0, temp = 0, mean = 0;
129             ;
130
131             //Berechnung der durschnittlichen Balkenbreite der 1. Ziffer
132             for (i = 1; i <= 9; i++) {
133                     mean += timer_value[i];
134             }
135             mean = mean / 9;
136
137             //Entscheidung schmaler / breiter Balken
138             for (timer_value_position = 1; timer_value_position <= anz_timer_value;
139                             timer_value_position++) {
140                     if (timer_value[timer_value_position] <= mean) {
141                             code[char_cnt] = (code[char_cnt] << 1) | 0; //schmaler Balken = 0
142                             temp++;
143
144                     } else {
145                             code[char_cnt] = (code[char_cnt] << 1) | 1; //breiter Balken = 1
146                             temp++;
147                     }
```

```
148                     if (temp == 9) {
149                             char_cnt++;
150                             timer_value_position++;
151                             temp = 0;
152
153                             //für jedes Zeichen neue Mittelwertwert bestimmen
154                             mean = 0;
155                             for (i = timer_value_position; i <= (timer_value_position + 9);
156                                             i++) {
157                                     mean += timer_value[i];
158                             }
159                             mean = mean / 9;
160                     }
161             }
162     }
163
164     //testen ob erstes und letztes Zeichen * ist
165     void check_code() {
166
167             if (code[0] == 148 && code[char_cnt - 1] == 148)
168                     decode = 1;
169             else {
170                     UARTprintf("Fehler ");
171                     decode = 0;
172             }
173     }
174
175     //umsetzen der Dezimalzahlen in Zeichen
176     void decode_to_char(void) {
177             int i = 0;
178             for (i = 1; i <= char_cnt - 2; i++) {
179                     switch (code[i]) {
180                     case 148:
181                             UARTprintf("*");
182                             break;
183
184                     case 52:
185                             UARTprintf("0");
```

```
186                 break;
187             case 289:
188                 UARTprintf("1");
189                 break;
190             case 97:
191                 UARTprintf("2");
192                 break;
193
194             case 352:
195                 UARTprintf("3");
196                 break;
197
198             case 49:
199                 UARTprintf("4");
200                 break;
201             case 304:
202                 UARTprintf("5");
203                 break;
204             case 112:
205                 UARTprintf("6");
206                 break;
207             case 37:
208                 UARTprintf("7");
209                 break;
210             case 292:
211                 UARTprintf("8");
212                 break;
213             case 100:
214                 UARTprintf("9");
215                 break;
216             case 265:
217                 UARTprintf("A");
218                 break;
219             case 73:
220                 UARTprintf("B");
221                 break;
222             case 328:
223                 UARTprintf("C");
```

```
224                 break;
225             case 25:
226                 UARTprintf("D");
227                 break;
228             case 280:
229                 UARTprintf("E");
230                 break;
231             case 88:
232                 UARTprintf("F");
233                 break;
234             case 13:
235                 UARTprintf("G");
236                 break;
237             case 268:
238                 UARTprintf("H");
239                 break;
240             case 76:
241                 UARTprintf("I");
242                 break;
243             case 28:
244                 UARTprintf("J");
245                 break;
246             case 259:
247                 UARTprintf("K");
248                 break;
249             case 67:
250                 UARTprintf("L");
251                 break;
252             case 322:
253                 UARTprintf("M");
254                 break;
255             case 19:
256                 UARTprintf("N");
257                 break;
258             case 274:
259                 UARTprintf("O");
260                 break;
261             case 82:
```

```
262                     UARTprintf("P");
263                     break;
264             case 7:
265                     UARTprintf("Q");
266                     break;
267             case 262:
268                     UARTprintf("R");
269                     break;
270             case 70:
271                     UARTprintf("S");
272                     break;
273             case 22:
274                     UARTprintf("T");
275                     break;
276             case 385:
277                     UARTprintf("U");
278                     break;
279             case 193:
280                     UARTprintf("V");
281                     break;
282             case 448:
283                     UARTprintf("W");
284                     break;
285             case 145:
286                     UARTprintf("X");
287                     break;
288             case 400:
289                     UARTprintf("Y");
290                     break;
291             case 208:
292                     UARTprintf("Z");
293                     break;
294             case 196:
295                     UARTprintf(" ");
296                     break;
297             case 168:
298                     UARTprintf("$");
299                     break;
```

```
300                     case 162:
301                             UARTprintf("/");
302                             break;
303                     case 138:
304                             UARTprintf("+");
305                             break;
306                     case 42:
307                             UARTprintf("%%");
308                             break;
309                     case 133:
310                             UARTprintf("-");
311                             break;
312                     case 388:
313                             UARTprintf(".");
314                             break;
315                     default:
316                             UARTprintf("NaN ");
317                     }
318             }
319     }
320     //#############################################################################
321     ###
322     int main(void) {
323             int i = 0;
324             SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN
325                                             | SYSCTL_XTAL_16MHZ); //16MHz Takt vom Quartz
326             uart_ini();
327             timer_ini();
328             interrupt_ini();
329             port_ini();
330
331             while (1) {
332                     //wenn 0,5 s kein Interrupt mehr aufgetreten ist
333                     //und weniger als 675 Flanken aufgetreten sind
334                     //-> decodierung starten
335                     if (TimerValueGet(TIMER2_BASE, TIMER_A) >= 8000000) {
336                             //wenn mind. ein Zeichen eingelesen ist
337                             if (anz_timer_value >= 9) {
```

```
338                                     if (full == 0) {
339                                             IntDisable(0); //Interrupt verbieten
340                                             GPIO_PORTD_DATA_R = 0xFF; //LED an
341
342                                             decode_to_array();
343                                             check_code();
344                                             if (decode == 1)
345                                                     decode_to_char();
346
347                                             UARTprintf("\n"); //Zeilenumbruch nach jedem Code
348
349                                             for (i = 0; i < 20; i++) {
350                                                     code[i] = 0;
351                                             }
352                                             char_cnt = 0;
353                                             anz_timer_value = 0;
354                                             decode = 0;
355
356                                             GPIO_PORTD_DATA_R = 0x00; //LED aus
357                                             IntEnable(0); //Interrupt erlauben
358                                     } else {
359                                             UARTprintf("Speicher voll");
360                                             anz_timer_value = 0;
361                                             full = 0;
362                                     }
363
364                                     //wenn weniger als ein Zeichen
365                                     //oder mehr als 675 Flanken eingelesen wurde
366                             } else {
367                                     anz_timer_value = 0;
368                                     SysCtlPeripheralReset(SYSCTL_PERIPH_TIMER2);
369                                     TimerConfigure(TIMER2_BASE,
370     TIMER_CFG_PERIODIC_UP);
371                                     TimerEnable(TIMER2_BASE, TIMER_BOTH);
372                             }
373                     }
374             }}
```

B Quellcodes: QR Code

```
1  /*-------------------------------------------------------------------------------
2   *  QR code Scanner
3   *
4   *  Torsten Persson       WS12/13
5   *-------------------------------------------------------------------------------
6   */
7  #define TARGET_IS_TEMPEST_RB1
8  #include "stdlib.h"
9  #include "string.h"
10 #include "inc/lm3s9b92.h"
11 #include "inc/hw_memmap.h"
12 #include "inc/hw_types.h"
13 #include "driverlib/debug.h"
14 #include "driverlib/gpio.h"
15 #include "driverlib/pin_map_LM3S9B92.h"
16 #include "driverlib/rom.h"
17 #include "driverlib/sysctl.h"
18 #include "utils/uartstdio.h"
19 #include "driverlib/uart.h"
20 #include "driverlib/systick.h"
21 #include "driverlib/interrupt.h"
22 #include "driverlib/timer.h"
23 #include "driverlib/pin_map.h"
24 #include "driverlib/rom.h"
25 #include "quirc_internal.h"
26 #include "quirc.h"
27
28 int start = 0;
29 uint8_t picture[MAXZEILEN][MAXSPALTEN];
30
31 #ifdef DEBUG//
32 //*****************************************************************************
33
```

```
34  //*****************************************************************************
35  //
36  // The error routine that is called if the driver library encounters an error.
37  //
38  void
39  __error__(char *pcFilename, unsigned long ulLine)
40  {
41  }
42  #endif
43
44  //ISR
45  void start_prog(void) {
46          ROM_GPIOPinIntClear(GPIO_PORTB_BASE, GPIO_PIN_4);
47          if (start == 0)
48                  start = 1;
49  }
50
51  void wait(void) {
52          volatile int tmp;
53          for (tmp = 0; tmp < 4000000; tmp++)
54                  ;
55  }
56
57  void uart_ini(void) {
58          ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0); // UART0 einschalten
59          ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA); // Port A freigeben
60          GPIOPinConfigure(GPIO_PA0_U0RX); // Pin 0 als Receive
61          GPIOPinConfigure(GPIO_PA1_U0TX); // Pin 1 als Transmit
62          // Port A Pin 0 und 1 für UART0
63          ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
64   // Baudrate: 115200
65          ROM_UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
66          (UART_CONFIG_WLEN_8|UART_CONFIG_STOP_ONE                                |
67  UART_CONFIG_PAR_NONE));
68
69          ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1); // UART1 einschalten
70          ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD); // Port D freigeben
71          GPIOPinConfigure(GPIO_PD2_U1RX); // Pin 2 als Receive
```

```
72              GPIOPinConfigure(GPIO_PD3_U1TX); // Pin 3 als Transmit
73              // Port D Pin 2 und 3 für UART0
74              ROM_GPIOPinTypeUART(GPIO_PORTD_BASE, GPIO_PIN_2 | GPIO_PIN_3);
75               // Baudrate: 19200
76              ROM_UARTConfigSetExpClk(UART1_BASE, SysCtlClockGet(), 19200,
77              (UART_CONFIG_WLEN_8|UART_CONFIG_STOP_ONE                                |
78  UART_CONFIG_PAR_NONE));
79  }
80
81  /* Interrupt for User Button on Port B Pin 4*/
82  void interrupt_ini(void) {
83              ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB); // Port B freigeben
84              ROM_GPIOPinTypeGPIOInput(GPIO_PORTB_BASE, GPIO_PIN_4); // Pin 4 als GPIO
85  setzen
86              // Interrupt an Pin 4 Port B bei fallender Flanke
87              ROM_GPIOIntTypeSet(GPIO_PORTB_BASE, GPIO_PIN_4, GPIO_FALLING_EDGE);
88              ROM_GPIOPinIntEnable(GPIO_PORTB_BASE,  GPIO_PIN_4);  //  Pin  4  als  Interrupt
89  zulassen
90              GPIOPortIntRegister(GPIO_PORTB_BASE, start_prog); // start_prog() als ISR setzen
91              IntMasterEnable();
92  }
93
94  //#############################################################################
95  ###
96
97  int main(void) {
98
99  //          Set the system clock to run at 80MHz from the PLL.
100             ROM_SysCtlClockSet(SYSCTL_SYSDIV_2_5        |        SYSCTL_USE_PLL        |
101 SYSCTL_OSC_MAIN
102                                             | SYSCTL_XTAL_16MHZ);
103
104             uart_ini();
105             interrupt_ini();
106             struct quirc qr;
107             char out_buf[20];
108             struct quirc_code code;
109             struct quirc_data data;
110             quirc_decode_error_t err;
111             qr.image = &picture[0][0];
```

```
112
113             while (1) {
114
115                     if (start == 1) {
116
117                             ROM_IntMasterDisable(); //Interrups ausschalten
118
119                             quirc_begin(&qr, MAXSPALTEN, MAXZEILEN);
120                             strcpy(out_buf, "Bild einlesen \n\r");
121                             uart_puts_pc(out_buf);
122
123                             /*Bild einlesen*/
124                             err = get_pic(&qr);
125                             if (err == 0) {
126                                     strcpy(out_buf, "Bild eingelesen \n\r");
127                                     uart_puts_pc(out_buf);
128                                     strcpy(out_buf, "Code suchen \n\r");
129                                     uart_puts_pc(out_buf);
130
131                                     /*Bild auswerten*/
132                                     quirc_end(&qr);
133                                     if (qr.num_grids == 1) {
134                                             strcpy(out_buf, "1 Code gefunden \n\r");
135                                             uart_puts_pc(out_buf);
136                                             quirc_extract(&qr, 0, &code);
137
138                                             /* Dekodieren */
139                                             err = quirc_decode(&code, &data);
140                                             if (err) {
141                                                     strcpy(out_buf, quirc_strerror(err));
142                                                     uart_puts_pc(out_buf);
143                                                     strcpy(out_buf, "\n\r");
144                                                     uart_puts_pc(out_buf);
145                                             } else {
146                                                     strcpy(out_buf, data.payload);
147                                                     uart_puts_pc(out_buf);
148                                                     strcpy(out_buf, "\n\r");
149                                                     uart_puts_pc(out_buf);
```

```
150                                         }
151                                 } else {
152                                         strcpy(out_buf, "kein Code\n\r");
153                                         uart_puts_pc(out_buf);
154                                 }
155                         }
156                         memset(picture, 0, sizeof(picture)); //Bild löschen
157                         ROM_IntMasterEnable(); //Interrupts freigeben
158                         start = 0;
159                 }
160         }
161 }
```

C Quellcode: camera.c

```
 1  /*-------------------------------------------------------------------------------------
 2   *  Kommunikation mit der Kamera
 3   *
 4   *  Torsten Persson       WS12/13
 5   *-------------------------------------------------------------------------------------
 6   */
 7  #include "inc/lm3s9b92.h"
 8  #include "inc/hw_memmap.h"
 9  #include "inc/hw_types.h"
10  #include "driverlib/debug.h"
11  #include "driverlib/gpio.h"
12  #include "driverlib/pin_map_LM3S9B92.h"
13  #include "driverlib/rom.h"
14  #include "driverlib/sysctl.h"
15  #include "stdlib.h"
16  #include "stdint.h"
17  #include "string.h"
18  #include "utils/uartstdio.h"
19  #include "driverlib/uart.h"
20  #include "quirc.h"
21  #include "quirc_internal.h"
22
23  #ifdef DEBUG//
24  //*****************************************************************************
25  //
26  // The error routine that is called if the driver library encounters an error.
27  //
28  void
29  __error__(char *pcFilename, unsigned long ulLine)
30  {
31  }
32  #endif
```

```
33
34  int StringLen = 0;
35  int zeile = 0, spalte = 0, i = 0;
36
37  //Liest die Antwort der Kamera
38  void uart_gets_cam(char* Buffer) {
39          char NextChar;
40          spalte = 0;
41          zeile = 0;
42
43          NextChar = UARTCharGet(UART1_BASE); // Warte auf und empfange das nächste
44  Zeichen
45
46          while (NextChar != ':') {
47                  *Buffer++ = NextChar;
48                  NextChar = UARTCharGet(UART1_BASE);
49          }
50          *Buffer = '\0';
51  }
52
53  //Liest die Daten die von der Kamera kommen ein
54  //s/w entscheidung
55  //speicher jedes Pixel als Byte
56  void uart_gets_cam_dat(struct quirc *q) {
57          uint8_t NextCharL = 0, NextCharH = 0;
58          uint16_t pixel = 0;
59
60          NextCharL = UARTCharGet(UART1_BASE); // Warte auf und empfange das nächste
61  Zeichen
62
63          while (spalte < q->w) {
64                  NextCharL = UARTCharGet(UART1_BASE);
65                  NextCharH = UARTCharGet(UART1_BASE);
66                  pixel = ((NextCharH << 8) + NextCharL); //zusammenführung der
67  Farbinformation
68                  //maskierung der einzelnen Farben und s/w Entscheidung
69                  if (((pixel & 0xF800) >> 11) >= 0x10 && ((pixel & 0x7E0) >> 5) >= 0x20
70                                  && (pixel & 0x1F) >= 0x10)
71                          *((q->image + spalte) + (zeile * 160)) = 0x00;
```

```
72                      else
73                              *((q->image + spalte) + (zeile * 160)) = 0x01;
74                      zeile++;
75                      StringLen++;
76
77                      if (zeile >= q->h) {
78                              spalte++;
79                              zeile = 0;
80                              //Daten noch nicht vollständig, DAT: wird eingelesen
81                              if (spalte != 160) {
82                                      for (i = 0; i <= 5; i++) {
83                                              NextCharL = UARTCharGet(UART1_BASE);
84                                      }
85                              } else
86                                      //Daten komplett, : wird eingelesen
87                                      UARTCharGet(UART1_BASE);
88                      }
89              }
90      }
91
92      //schickt Befehle an die Kamera
93      void uart_puts_cam(char *s) {
94              while (*s) // so lange *s != '\0'
95              {
96                      UARTCharPut(UART1_BASE, *s);
97                      s++;
98              }
99              UARTCharPut(UART1_BASE, 0x0D); //Kennzeichung vom Ende des Befehls
100     }
101
102     //schickt daten an PC
103     void uart_puts_pc(char *s) {
104             while (*s) //so lange *s != '\0'
105             {
106                     UARTCharPut(UART0_BASE, *s);
107                     s++;
108             }
109     }
```

```
110
111 //erstellt eine Bitmapdatei aus den eingelesen Daten
112 void make_bmp(struct quirc *q) {
113
114         //Array mit eingetragenem Header
115         uint8_t bmp_array[2462] = { 0x42, 0x4d, 0x09, 0x96, 0x00, 0x00, 0x00, 0x00,
116                         0x00, 0x00, 0x3E, 0x00, 0x00, 0x00, 0x28, 0x00, 0x00, 0x00, 0xa0,
117                         0x00, 0x00, 0x00, 0x78, 0x00, 0x00, 0x00, 0x01, 0x00, 0x01, 0x00,
118                         0x00, 0x00, 0x00, 0x00, 0x09, 0x60, 0x00, 0x00, 0x00, 0x00, 0x00,
119                         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
120                         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0x00 };
121
122         int i = 62; //Daten hinter den Header einfügen
123         int temp = 0;
124
125         //Zeilen werden von oben nach unten ins Array gespeichert
126         for (zeile = MAXZEILEN - 1; zeile >= 0; zeile--) {
127                 for (spalte = 0; spalte < MAXSPALTEN; spalte++) {
128                         if (*((q->image + spalte) + (zeile * 160)) == 0) {
129                                 bmp_array[i] = (bmp_array[i] << 1) | 1; //Pixel wird invertiert
130                                 temp++;
131                         } else if (*((q->image + spalte) + (zeile * 160)) == 1) {
132                                 bmp_array[i] = (bmp_array[i] << 1) | 0; //Pixel wird invertiert
133                                 temp++;
134                         }
135                         if (temp == 8) {
136                                 i++;
137                                 temp = 0;
138                         }
139                 }
140         }
141         //Bitmap an den PC senden
142         for (i = 0; i < 2462; i++) {
143                 UARTCharPut(UART0_BASE, bmp_array[i]);
144         }
145         memset(bmp_array, 0, sizeof(bmp_array));
146 }
147
```

```
148
149 quirc_decode_error_t get_pic(struct quirc *q) {
150         char buffer[20];
151         StringLen = 0;
152         memset(buffer, 0, sizeof(buffer));
153
154 // reset
155         strcpy(buffer, "rs");
156         uart_puts_cam(buffer);
157         uart_gets_cam(buffer);
158         memset(buffer, 0, sizeof(buffer));
159
160         // schwarz/weiß Modus an
161         strcpy(buffer, "bw 1");
162         uart_puts_cam(buffer);
163         uart_gets_cam(buffer);
164         if (strcmp(buffer, "ACK\r") != 0) {
165                 strcpy(buffer, "bw Fehler ");
166                 uart_puts_pc(buffer);
167                 return CAMERA_COMMAND_ERROR;
168         }
169         memset(buffer, 0, sizeof(buffer));
170
171         // Poll modus an
172         strcpy(buffer, "pm 1");
173         uart_puts_cam(buffer);
174         uart_gets_cam(buffer);
175         if (strcmp(buffer, "ACK\r") != 0) {
176                 strcpy(buffer, "pm Fehler ");
177                 uart_puts_pc(buffer);
178                 return CAMERA_COMMAND_ERROR;
179         }
180         memset(buffer, 0, sizeof(buffer));
181
182         // LED aus
183         strcpy(buffer, "L0");
184         uart_puts_cam(buffer);
185         uart_gets_cam(buffer);
```

```
186             if (strcmp(buffer, "ACK\r") != 0) {
187                     strcpy(buffer, "LED Fehler ");
188                     uart_puts_pc(buffer);
189                     return CAMERA_COMMAND_ERROR;
190             }
191             memset(buffer, 0, sizeof(buffer));
192
193             // Daten anfordern
194             strcpy(buffer, "sf 2 2");
195             uart_puts_cam(buffer);
196             uart_gets_cam(buffer);
197             if (strcmp(buffer, "ACK\rDAT") != 0) {
198                     strcpy(buffer, "sf Fehler ");
199                     uart_puts_pc(buffer);
200                     return CAMERA_COMMAND_ERROR;
201             }
202             uart_gets_cam_dat(q);
203             make_bmp(q);
204             return QUIRC_SUCCESS;
205     }
```

D Quellcode: Bitmap aus Logfile generieren

```
1  /*-----------------------------------------------------------------------------
2   *  Erstellt ein Bitmapfile aus dem Logfile von HTerm
3   *
4   *  Torsten Persson       WS12/13
5   *-----------------------------------------------------------------------------
6   */
7  #include <stdio.h>
8  #include <string.h>
9
10 int main() {
11         char datenArray[2462];
12
13         FILE*myStream_read=fopen(""C:\\Documentsand Settings\\test\\Desktop\\output"","rb");
14         FILE*myStream_write=fopen(""C:\\Documents and Settings\\test\\Desktop\\code.bmp",
15 "w");
16         if (myStream_read==NULL){
17                 printf("Output Datei nicht gefunden\n");
18         }
19         if (myStream_write==NULL){
20                 printf("Code.bmp konnte nicht erstellt werden\n");
21         }
22
23         fseek(myStream_read,16L,SEEK_SET);
24         fread(datenArray, sizeof(char), 2462, myStream_read);
25         fwrite(datenArray, sizeof(char), 2462, myStream_write);
26
27         fclose(myStream_read);
28         fclose(myStream_write);
29
30         system("PAUSE");
31 return 0;
32 }
```

E Quelltext: Formatinformation einlesen

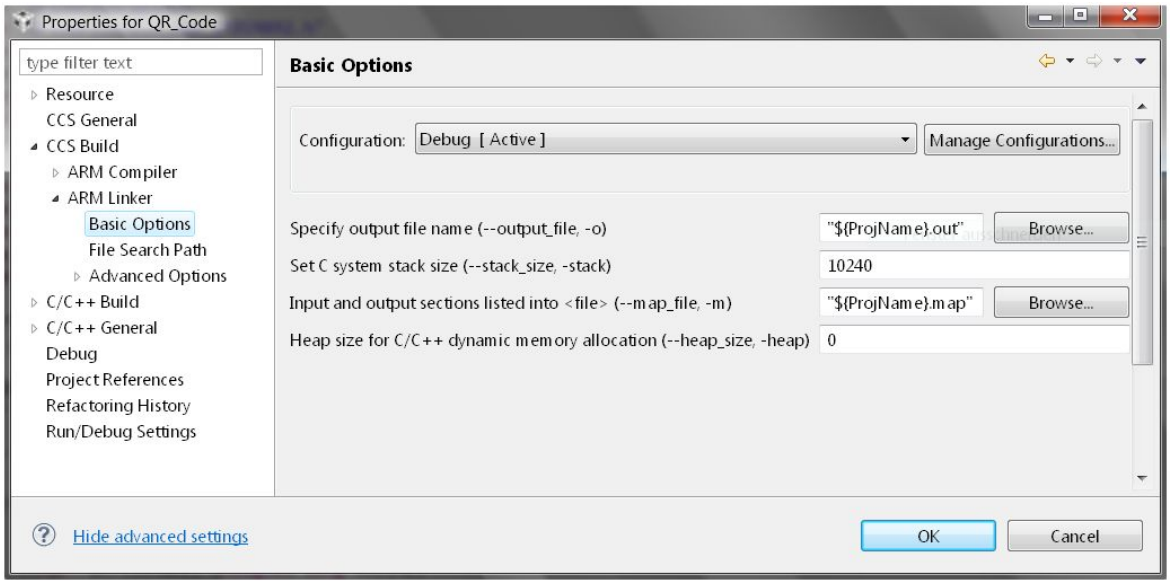
```
1   //gehört zu decode.c
2   //geändert durch Torsten Persson
3   //hinzufügen der Dekodierung mithilfe der Tabelle
4   static quirc_decode_error_t read_format(const struct quirc_code *code,struct quirc_data *data,
5    int which)
6   {
7           int i,j;
8           uint16_t format = 0,check=0,check_sum=0;
9           uint16_t fdata,x;
10          quirc_decode_error_t err;
11          if (which) {
12                  //Bei Eckstein A
13                  for (i = 0; i < 7; i++)
14                          format = (format << 1) |
15                                  grid_bit(code, 8, code->size - 1 - i);
16                  //Bei Eckstein C
17                  for (i = 0; i < 8; i++)
18                          format = (format << 1) |
19                                  grid_bit(code, code->size - 8 + i, 8);
20          // Formatinformation bei Eckstein B
21          } else {
22                  static const int xs[15] = {
23                          8, 8, 8, 8, 8, 8, 8, 8, 7, 5, 4, 3, 2, 1, 0
24                  };
25                  static const int ys[15] = {
26                          0, 1, 2, 3, 4, 5, 7, 8, 8, 8, 8, 8, 8, 8, 8
27                  };
28                  for (i = 14; i >= 0; i--)
29                          format = (format << 1) | grid_bit(code, xs[i], ys[i]);
30          }
31
32
33          //Formatinformation anhand der Lookup table auswerten
```

```
34              format ^= 0x5412;         //Maskierung lösen
35              err = QUIRC_ERROR_FORMAT_ECC;
36              for(i=0; I <= 31; i++){
37                      check = format^format_lookup[i];
38                      check_sum = 0;
39                      for (j=0; j <= 14; j++){
40                              x= 0x4000 >> j;
41                              if ((check & x) == x){
42                                      check_sum = (check_sum<<1)|1;
43                              }
44                      }
45                      if (check_sum <= 7){
46                              format = format_lookup[i];
47                              err = QUIRC_SUCCESS;
48                              break;
49                      }
50              }
51              if (err)
52                      return err;
53              fdata = format >> 10;
54              data->ecc_level = fdata >> 3;
55              data->mask = fdata & 7;
56              return QUIRC_SUCCESS;
57      }
```

F Hinweis zum Einstellen der Größe des Stacks

Die dafür vorgesehene Einstellung befindet sich im Code Composer Studio v5 unter:

Projects→Properties:



In dem Feld „Set C system stack size“ kann die gewünschte Größe in Byte angegeben werden. Standardmäßig ist er auf 256 Byte gesetzt.

G Auszug aus dem Linkerfile

```
******************************************************************************
               TMS470 Linker PC v4.9.1
******************************************************************************
>> Linked Thu Jan 03 18:57:26 2013

OUTPUT FILE NAME:   <QR_Code.out>
ENTRY POINT SYMBOL: "_c_int00"  address: 0000538d

MEMORY CONFIGURATION

       name            origin    length      used     unused   attr    fill
----------------------  --------  ---------  --------  --------  ----  --------
  FLASH                00000000   00040000  00006dd6  0003922a  R  X
  SRAM                 20000000   00018000  00007adc  00010524  RW X

SEGMENT ALLOCATION MAP

run origin  load origin   length   init length attrs members
----------  ----------- ---------- ----------- ----- -------
00000000    00000000    00006de0   00006de0    r-x
  00000000    00000000    000055c2   000055c2    r-x .text
  000055c8    000055c8    00001744   00001744    r-- .const
  00006d10    00006d10    000000d0   000000d0    r-- .cinit
20000000    20000000    000077fc   00000000    rw-
  20000000    20000000    0000026c   00000000    rw- .vtable
  2000026c    2000026c    00004d90   00000000    rw- .bss
  20004ffc    20004ffc    00002800   00000000    rw- .stack
200077fc    200077fc    000002e4   000002d8    rw-
  200077fc    200077fc    000002d8   000002d8    rw- .data
  20007ad8    20007ad8    00000008   00000000    rw- .sysmem
```

Eidesstattliche Erklärung

Ich versichere, dass ich vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Hamburg, den 04. Januar 2013 Unterschrift: