Otto-von-Guericke-Universität Magdeburg

Institut für Verteilte Systeme
AG Eingebettete Systeme und Betriebssysteme

Entwicklung eines Bootloader für über CAN verbundene Mikrocontroller

Laborpraktikumsbericht

Jörg Diederich

Wintersemester 2006

Betreuer:

Prof. Jörg Kaiser
Dipl. Inform. Michael Schulze

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Motivation und Aufgabenstellung

Die Softwareentwicklung für Mikrocontroller lässt sich in zwei Phasen unterteilen. In der ersten Phase, bis zum Abschluss der Implementierung, wird auf einem Fremdsystem entwickelt. Durch ihre weite Verbreitung kommen zumeist Personalcomputer zum Einsatz. Mit Verwendung von Crosscompilern lässt sich Programmcode für den gewählten Mikrocontroller erzeugen. Erst mit der Übertragung des Programmcode auf den Mikrocontroller beginnt die zweite Phase der Softwareentwicklung, nun auf dem beabsichtigtem Zielsystem.
Durch die Fähigkeit von Mikrocontrollern zum sog. In-System-Programming wird der Übergang zwischen beiden Entwicklungsphasen vereinfacht. Auf aufwendige Umbaumaßnahmen der Hardware oder gar einem Ausbau des Mikrocontrollers aus einem kompletten System kann verzichtet werden. Vielmehr kann ein Mikrocontroller, wie der Name schon vermuten lässt, beim Übertragen von Programmcode im System verbleiben. Dies reduziert die mechanische Beanspruchung des gesamten Systems, zudem wird viel Zeit eingespart. Die gewonnene Flexibilität lässt sich beispielsweise für zusätzliche Tests im Rahmen der Softwareentwicklung einsetzen. Das In-System-Programming (im Folgendem auch kurz mit ISP bezeichnet) nutzt vorhandene Kommunikationsmöglichkeiten eines Mikrocontrollers. Bekannt sind serielle Möglichkeiten wie JTAG oder SPI. Ein entsprechendes Kabel samt Adapter stellt dabei die Verbindung zwischen PC und Mikrocontroller her.
Mikrocontroller der AT90CAN-Baureihe der Firma Atmel besitzen eine Schnittstelle für den CAN-Bus. Wird diese Schnittstelle als vornehmliche Kommunikationsmöglichkeit verwendet, erscheint die Verwendung einer zusätzlichen Schnittstelle lediglich zum Zwecke des ISP überflüssig.
Die Möglichkeit zum ISP über den existierenden CAN-Busanschluss bietet mehrere Vorteile:

- Es entfällt das Anschließen eines zusätzlichen Verbindungskabels, wenn ausschließlich per CAN kommuniziert wird. Zusätzlich bleibt in diesem Falle eine Schnittstelle des Mikrocontrollers ständig für andere Aufgaben frei.
- Es entfällt der oft lästige Wechsel des jeweils angeschlossenen Mikrocontrollers. Bisherige Verfahren erlauben nur die Kommunikation mit einem einzelnen Mikrocontroller. Für die Verwendung mehrerer Mikrocontroller muss bisher immer das Verbindungskabel getrennt und neu angeschlossen werden.
- Es wird zusätzlich Zeit eingespart, da mehrere Mikrocontroller gleichzeitig mit neuem Programmcode versehen werden können. Gerade bei Verwendung mehrerer Mikrocontroller nimmt das bisher praktizierte sequentielle Vorgehen viel Zeit in Anspruch.
- Letztendlich wird auch die mechanische Beanspruchung der Anschlüsse reduziert.

Ziel des durchgeführten Laborpraktikums war es, die genannten Vorteile durch eine entsprechende Entwicklung nutzbar zu machen. Grundlage waren, wie bereits angedeutet, Mikrocontroller des Typs Atmel AT90CAN128. Die Aufgabe umfasste Konzeption eines Befehlprotokolls und entsprechende Programmierung der Mikrocontroller. Weiterhin galt es, einen entsprechenden Gegenpart auf Seite eines PC zu entwickeln. Als Resultat

sollte der eingangs beschriebene Prozess der Softwareentwicklung für Mikrocontroller weiterhin durchführbar sein.

2. Anforderungen

Die Anforderungen lassen sich in mehrere Bestandteile zerlegen. Zunächst müssen die Eigenschaften der verwendeten Kommunikation beachtet werden. Weiterhin stellt der zu verwendende Mikrocontroller Anforderungen. Schlussendlich sind die Voraussetzungen zur Softwareentwicklung und die Anforderungen an die Funktionalität der Lösung zu berücksichtigen.

2.1. Controller Area Network CAN

Der Begriff CAN bezeichnet einen seriellen Feldbus. Für auf dem Bus versandte Nachrichten sind verschiedene Formate spezifiziert. Für den unmittelbaren Datenaustausch interessant sind der sog. Daten-Frame und der Remote-Frame. Diese setzen sich im Wesentlichen aus folgenden Bestandteilen zusammen:

1. Bezeichner (Identifier)
2. Anzahl der Datenbytes (Data Length Code)
3. Datenbytes (Data)

Vervollständigt wird die Nachricht durch verschiedene Bits und durch eine Checksumme. Andere spezifizierte Formate dienen der Steuerung des Datenflusses.
Die Kommunikation über CAN basiert nicht auf der Adressierung des Empfängers, sondern auf der Kennzeichnung einer Nachricht [5]. Die aktuelle CAN-Spezifikation CAN 2.0 sieht für den Bezeichner einer Nachricht zwei verschiedene Längen vor. CAN 2.0A spezifiziert einen sog. Standard-Identifier von 11 Bit Länge. In CAN 2.0B wird der Identifier auf 29 Bit erweitert, in diesem Fall wird von einem Extended-Identifier gesprochen. Über die Spezifikationen hinweg gleich bleibt die mögliche Anzahl an Datenbytes: Nachrichten im CAN-Bus enthalten maximal 8 Byte Daten. Auch Nachrichten ohne Datenbytes sind zulässig. Die Anzahl enthaltener Datenbytes wird durch das entsprechende Feld in einer Nachricht beschrieben.
Die CAN-Spezifikation ist unabhängig vom verwendeten Übertragungsmedium. Verfügbar sind drahtgebundene und drahtlose bzw. optische Lösungen [4]. Aufgrund dieser Unabhängigkeit variiert der spezifizierte maximale Durchsatz. In der gebräuchlichen Spezifizierung nach *ISO 11898-2 high speed*, welche die Verwendung von Twisted-Pair-Kabel vorsieht, sind maximal 1 Mbit/s Durchsatz definiert.
Die Besonderheit des CAN-Busses ist die Arbitrierung nach CSMA/CA. Um Kollisionen im Falle eines gleichzeitigen Buszugriffs zu vermeiden, wird sich des Bezeichners einer Nachricht bedient. Stellt ein Sender anhand des Bezeichners fest, dass seine Nachricht niedriger priorisiert ist als die Nachricht eines gleichzeitig sendenden anderen Teilnehmers, stellt er seinen Sendevorgang ein. Erst anschließend kann dieser Teilnehmer einen erneuten Sendeversuch durchführen. Auf diesem Wege wird der Buszugriff gesteuert und zudem der Versand der Nachricht mit dem höchstpriorisiertem Bezeichner sichergestellt. Im Falle des gleichzeitigen Versands von Nachrichten mit gleichen Bezeichnern liegt ein Fehler vor. Dieser Zustand darf nicht eintreten, kann aber von den Teilnehmern erkannt und entsprechend behandelt werden.

2.2. Atmel AT90CAN128 Mikrocontroller

Der Programmspeicher des AT90CAN128 besteht aus den Bereichen Flash, EEPROM sowie mehreren dedizierten Fuse-Bytes. Für das Schreiben von Daten in diese Bereiche existieren nach [1] mehrere Möglichkeiten. Von der Hardware direkt unterstützt wird die Programmierung per JTAG oder im parallelem Modus. Ebenfalls ohne zusätzlichen Aufwand möglich ist die Programmierung via SPI.
Zusätzlich lassen sich die genannten Möglichkeiten flexibel erweitern. Zu diesem Zweck stellt der Mikrocontroller den sog. Boot Loader Support bereit. Er erlaubt es, individuell angepassten Programmcode zum Beschreiben des Programmspeichers zu verwenden. Für den Boot Loader Support stellt der Mikrocontroller einen eigenen Speicherbereich innerhalb des Flash bereit, die sog. Boot Loader Flash Section. In der maximalen Konfiguration umfasst dieser Bereich 8 Kilobyte. Aus diesem Bereich ausgeführter Programmcode hat in der Voreinstellung vollen Zugriff auf alle Ressourcen des Mikrocontrollers. Dies umfasst sowohl den Arbeitsspeicher von 4 Kilobyte, als auch die Speicherbereiche Flash und EEPROM. Gleichfalls bleibt die Nutzung von Interrupts möglich. Mit entsprechend gesetzten Bits kann die direkte Ausführung von Programmcode aus der Boot Loader Flash Section nach Start oder Reset des Mikrocontrollers erreicht werden. So lässt sich eine erneute Ausführung als Reaktion auf eine externe Bedingung sicherstellen.
Der im AT90CAN128 verbaute CAN-Controller reklamiert die Kompatibilität zu den Spezifikationen CAN 2.0A und 2.0B. Nachrichten können sowohl über Polling als auch unter Verwendung von Interrupts gesendet und empfangen werden. Weiterhin besteht die Möglichkeit, Masken für den Empfang von Nachrichten vorzugeben. Mit diesen lässt sich bereits durch die Hardware ein Akzeptanzfilter für Nachrichten auf dem CAN-Bus einrichten. Eine Beanspruchung des Mikrocontrollers durch nicht interessierende Nachrichten wird damit verhindert.

2.3. Voraussetzungen der Softwareentwicklung

Auf Seite des Benutzers soll die zu entwickelnde Lösung unter dem Betriebssystem Linux lauffähig sein. Die Entwicklung von Software für den Betrieb auf dem Mikrocontroller bleibt davon unberührt.
Als Schnittstelle zwischen PC und CAN-Bus dienen Produkte der Firma PEAK-System. Unter Linux existieren durch den Hersteller bereits Treiber für diese Produkte. Vorrätig und zu unterstützen sind die sog. PCAN-Dongles und die PCAN-PCI Einsteckkarte. Für diesen Zweck kann auf die *libpcan*-Bibliothek zurückgegriffen werden, die zusammen mit den Treibern erhältlich ist.

2.4. Funktionalität der Lösung

Die zu entwickelnde Lösung soll den bekannten ISP-Prozess über ein CAN-Bussystem für mehrere AT90CAN128 realisieren. Dies beinhaltet Löschen, Schreiben und Lesen von Speicher. Als Speicherbereiche vorzusehen sind die Bereiche Flash und EEPROM. Nach Abschluss eines ISP-Prozesses soll eine eventuell auf dem Mikrocontroller gespeicherte Applikation zur Ausführung kommen.

Für den Betrieb sind zwei Möglichkeiten zur Nutzung vorzusehen. Nach einem Reset eines Mikrocontrollers soll das Durchführen des ISP-Prozesses erneut möglich sein. Dabei ist eine noch zu spezifizierende Bedingung einzuhalten, anhand derer alternativ die Applikation zur Ausführung kommt. Weiterhin soll ein Start aus einer Applikation heraus möglich sein, um direkt und ohne Reset ein erneutes Programmieren zu ermöglichen.

3. Existierende Lösungen

Es existieren bereits Lösungen für ähnliche Aufgabenstellungen. Auf diese lohnt ein Blick, um die Lösungsfindung zu beschleunigen.

3.1. Softwareentwicklung für den AT90CAN128 Mikrocontroller

Zur Entwicklung von Software für den AT90CAN128 existieren mehrere Lösungen. Die Software auf Benutzerseite soll, wie in Abschnitt 2.3 beschrieben, unter Linux laufen. Um die gesamte Entwicklung einheitlich und den Aufwand gering zu halten, bietet es sich an, Lösungen für dieses Betriebssystem zu finden.
Mit dem *avr-gcc* besteht ein Port des Linux-Standardcompilers GCC [8]. Damit können die von der Softwareentwicklung unter Linux bekannten und bewährten Verfahren weiter angewandt werden. Als Beispiel hervorzuheben ist die durch *make* mögliche Automatisierung des Übersetzens. Als Programmiersprachen ergeben sich entsprechend C sowie C++. Das Einbinden von Assembler-Code ist über entsprechende Sprachkonstrukte möglich.
Als Pendant zur C-Standardbibliothek existiert eine vom Projekt *AVR Libc* erstellte gleichnamige Bibliothek [6]. Diese beinhaltet neben C-Standardroutinen bereits spezielle, an die verschiedenen Mikrocontroller der AVR-Familie angepasste Routinen. Beispielhaft zu nennen sind die Möglichkeiten zur Definition von Interrupt Service Routinen oder die allgegenwärtigen Zugriffe auf Register. Auch für in C++ geschriebene Programme ist die Anwendung gegeben.

3.2. Atmel CAN Bootloader

Vom Fabrikanten des AT90CAN128 existieren zwei Entwicklungen, welche es ermöglichen, Programmcode über die CAN-Schnittstelle an den Mikrocontroller zu übertragen.
Mit dem *CAN & UART Boot Loader* [2] wird eine Lösung für gleich zwei unterschiedliche Schnittstellen präsentiert. Kostenlos erhältlich, liegen sowohl eine ausführliche Dokumentation als auch der Quellcode vor. Für dessen Übersetzung wird die Verwendung eines Compilers der Firma IAR [9] vorausgesetzt. Dieser ist jedoch nur für Windows-Betriebssysteme erhältlich. Ein beiliegendes vorcompiliertes Paket kann nicht auf den eingesetzten Boards verwendet werden. Grund ist ein Fehler dieser Boards, der einen Einschub zweier zusätzlicher Befehle vor Inbetriebnahme der CAN-Schnittstelle erfordert.
Die Entwicklung *"Slim" CAN Boot Loader* [3] baut auf erstgenannter Lösung auf. Sie verwendet das gleiche Protokoll und verzichtet dabei lediglich auf die Unterstützung der UART-Schnittstelle. Wesentlich ist die Möglichkeit der Verwendung des *avr-gcc* als Übersetzer. Damit kann diese Entwicklung sowohl theoretisch als auch in der Anwendung untersucht werden.
Beide Lösungen sehen folgenden Ablauf beim Übertragen von Programmcode vor:

1. Öffnen
   Damit wird der angesprochene Mikrocontroller für den Empfang weiterer Befehle aktiviert. Eine dedizierte Auswahl oder die Auswahl aller Mikrocontroller ist möglich.

2. Auswahl des gewünschten Speicherbereichs
   Bei Verwendung von Speicherblöcken mit mehr als 64 Kilobyte muss die explizite Auswahl einer Seite (Page) vorgenommen werden.

3. Durchführen der Aktion auf dem Speicher
   Dies kann Löschen, Schreiben oder Lesen umfassen.

4. Schließen
   Dies beendet die Verbindung.

Alternativ kann das Schließen der Verbindung auch durch einen Befehl zum Neustart erfolgen.
Das Vorgehen wurde als strukturiert und durchdacht bewertet. Jedoch lässt sich das verwendete Protokoll nicht für den simultanen Betrieb mehrerer Mikrocontroller einsetzen:

> Für die Übertragung via CAN werden die von einem Mikrocontroller auszuführenden Aktionen im Bezeichner einer CAN-Nachricht abgelegt. Bei der Kommunikation zwischen lediglich zwei Teilnehmern stellt dies kein Problem dar. Besteht jedoch eine 1:n-Kommunikation, ist das Vorgehen im CAN-Bus nicht aufrecht zu erhalten. Mehrere Teilnehmer können in diesem Fall potentiell zeitgleich identische Bezeichner auf den Bus legen. Wie in Abschnitt 2.1 erläutert, führt dies zu Fehlern.
> Selbst wenn dieser Fall ausgeschlossen oder der im CAN vorgesehenen Korrektur überlassen wird, ist das Protokoll nicht anwendbar. Die Ursache ist in dem Umstand zu finden, dass gleiche Bezeichner sowohl für das Senden als auch das Bestätigen von Nachrichten benutzt werden. Da alle Teilnehmer sämtliche Nachrichten auf dem Bus hören, kann eine Einordnung der Nachrichten in Aktion oder Reaktion nicht immer korrekt getätigt werden. Deutlich zeigen lässt sich dies am Versand von Datenbytes zum Schreiben des Programmspeichers. Diese mit *ID_PROG_DATA* bezeichnete Nachricht wird unter gleichem Bezeichner mit genau einem Byte Anhang bestätigt. Für andere Busteilnehmer, welche ebenfalls auf Datenbytes warten, stellt die Bestätigung einer gültige *ID_PROG_DATA*-Nachricht dar. Haben sich alle Empfänger vorher im gleichem Zustand befunden, laufen sie von nun an auseinander.

Nach Einblick in Dokumentation und Quellcode bieten beide Lösungen alle für das ISP benötigten Funktionen an. Als zu verwendende Programmiersoftware gibt Atmel die Eigenentwicklung *FLIP* in der Version 2.4.4 an. Für das zu verwendende Betriebssystem Linux ist jedoch maximal Version 1.8.8 erhältlich. Ein Test der Version 2.4.6 unter Windows machte deutlich, dass der AT90CAN128 lediglich auf der Kommandozeile durch das Programm *batchIsp* unterstützt wird. Trotz der benutzerunfreundlichen Kommandozeile unter Windows konnte die Funktionalität der zweiten Lösung überprüft und bestätigt werden. Auch konnte gezeigt werden, dass bereits mit dem Anschluss eines zweiten Mikrocontrollers ein nichtdeterministisches Verhalten entsteht. Da *batchIsp* nicht auf einen zweiten Kommunikationspartner vorbereitet ist, gerät sein Ablauf völlig durcheinander. Bis zum Auftreten des genannten, theoretischen Fehlers der Befehlsübermittlung kommt es nicht.

3.3. Programmiersoftware Avrdude

Wie in Abschnitt 1 erläutert, erfolgt im Anschluss an eine Implementierung die Übertragung des Programmcode an den Mikrocontroller. Dafür verwendet werden Programme, die unter dem Begriff Programmiersoftware zusammengefasst werden können. Im eigentlichen Sinne programmieren diese Programme einen Mikrocontroller nicht, sondern kümmern sich lediglich um die Datenübertragung. Jedoch sind diese Programme für den Benutzer die einzig sichtbare Instanz im Programmierzyklus, womit die von der Tätigkeit abweichende Bezeichnung erklärt werden kann.
Das bereits in Abschnitt 3.2 genannte Programm *FLIP* bzw. *batchIsp* läuft lediglich unter Windows und wird daher nicht weiter betrachtet. Mit *Avrdude* existiert eine Programmiersoftware zur Verwendung auch unter Linux [7]. Für die in Abschnitt 3.1 genannten Anforderungen stellt dieses Programm den Quasi-Standard dar.
Von *Avrdude* unterstützt wird bisher das Programmieren über serielle und parallele Schnittstellen eines PC. Für diese Schnittstellen wird die Kommunikation über eine ganze Reihe sog. Programmieradapter unterstützt. Dabei handelt es sich um Adapter, welche individuell die Signale zwischen Software und Mikrocontroller umsetzen. Die Bedienung von *Avrdude* erfolgt über die Kommandozeile. Einem Aufruf aus Makefiles steht so nichts im Wege. Nicht unterstützt werden bisher Schnittstellen zu Bussystemen wie CAN. Auf eine simultane Kommunikation mit mehreren Partnern wird daher in *Avrdude* bisher verzichtet.
Da *Avrdude* unter der GPLv2 steht, ist der Quellcode verfügbar. Somit ist ein Einblick in die Architektur möglich, und es lassen sich ggf. eigene Modifikationen durchführen.

4. Entwurf

Vom Vorgehen erinnert das Programmieren eines Mikrocontrollers an ein Client-Server-System. Die Programmiersoftware als Client verlangt mit einem Request die Verarbeitung übermittelter Daten. Der Server kann dabei der Mikrocontroller selbst oder eine auf ihm laufende Software sein. Nach Abschluss der Bearbeitung antwortet der Server der Anfrage entsprechend in einem Response. Für eine erfolgreiche Kommunikation müssen sich beide Teilnehmer an festgelegte, gemeinsame Regeln halten. Die Regeln legt ein Protokoll fest.
Da der AT90CAN128 keine direkte Unterstützung für eine Programmierung über die CAN-Schnittstelle vorsieht, muss vom Boot Loader Support Gebrauch gemacht werden. Das zum Einsatz kommende Programm lässt sich, unabhängig von der Bezeichnung des Supports, als Bootloader einstufen: Es startet nach einem Reset, in Abhängigkeit von Bedingungen, das eigentliche System.
In den folgenden Abschnitten wird der Entwurf aller angesprochenen Elemente der entwickelten Lösung erläutert.

4.1. Befehlprotokoll

Die Entwicklung des Protokolls orientierte sich an dem in Abschnitt 3.2 vorgestellten Protokoll. Die dort aufgezeigte Problematik bei der Kommunikation mit mehreren Teilnehmern galt es jedoch zu beachten.
Zwingend notwendig ist die Individualisierung der Nachrichten eines jeden Teilnehmers. Nur so sind Zuordnungen möglich, die zur Erhaltung eines konsistenten Zustandes benötigt werden. Das CAN-Protokoll bietet mit dem Bezeichner einer Nachricht bereits eine derartige Möglichkeit. Dies widerspricht leider dem eigentlichen Gedanken von CAN, wonach der Bezeichner eine Nachricht und nicht den Sender kennzeichnet. Bei Anwendung eindeutiger Bezeichner für jeden Teilnehmer kann so aber die Trennung vom verwendeten Übertragungsprotokoll durchgeführt werden.
Abbildung 1 skizziert den Aufbau einer Nachricht zwischen Bootloader und Programmiersoftware. Jeder Teilnehmer sendet alle seine Nachrichten unter seinem Bezeich-

Abbildung 1: Grundsätzlicher Aufbau einer Nachricht zwischen Bootloader und Programmiersoftware

ner (Identifier). Die Eindeutigkeit der Identifier aller Teilnehmer ist durch den Anwender sicherzustellen.
Die anderen beiden Elemente der Nachricht bedingen einander. Das mit *Action* bezeichnete Feld enthält eine übertragene Aktion, das nachfolgende Feld entsprechend Zusatzinformationen zu einer Aktion. In Anlehnung an [2] sieht der Entwurf die folgenden Aktionen vor:

- Öffnen

- Schließen
- Auswahl eines Speichers
- Auswahl einer Adresse
- Löschen eines Speichers
- Schreiben in einen Speicher
- Lesen aus einem Speicher

Die Programmiersoftware, im Folgendem auch kurz als Host bezeichnet, ist der aktive Kommunikationspartner. Der Bootloader reagiert lediglich auf die empfangenen Nachrichten. Die Kommunikation läuft nach dem Request-Response-Verfahren ab. Auf jeden Request vom Host erfolgt mindestens ein Response vom Bootloader. Die Inhalte der Request-Nachrichten und ihrer entsprechenden Response-Nachrichten sind im Einzelnen in Anhang A aufgelistet.
Entsprechend den genannten Aktionen, ist folgender Programmablauf vorgesehen:

1. Aktivieren der Bootloader
   Dieser initialen Aktion kommt besondere Bedeutung zu. In einem Bussystem muss der Bootloader sicherstellen, dass sämtliche nachfolgende Befehle von genau einem Host stammen. Zu diesem Zweck vermerkt ein Bootloader den aktivierenden Sender, und wird bis zum Deaktivieren nur noch Befehle dieses Hosts akzeptieren.
   Im Einsatz mit mehreren Bootloadern kann der Umstand eintreten, dass einige aktiviert sind, während andere auf eine Aktivierung warten. Um eine störungsfreie Kommunikation zwischen Host und aktivierten Bootloadern zu gewährleisten, ignorieren nicht aktivierte Bootloader alle Aktionen bis auf diese.

2. Auswahl des Speichers
   Mikrocontroller können potentiell mehrere Speicher beinhalten, beispielsweise Flash- oder EEPROM-Speicher. Adressen können sich zwischen verschiedenen Speicherbereichen überlappen. Daher wird vor nachfolgenden Aktionen eine Auswahl notwendig. Voreingestellt ist der Bereich des Flash-Speichers.

3. Auswahl der Adresse
   Nachfolgende Aktionen können einzelne Adressbereiche eines Speicherbereichs betreffen. In diesem Falle ist eine vorherige Angabe notwendig. Voreingestellt ist Adresse 0.

4. Ausführen des ISP
   Waren sämtliche Aktionen bisher vorbereitend, wird nun das eigentliche ISP durchgeführt. Sollen Daten geschrieben werden, erfolgt das Übertragen der Daten vom Host an die Bootloader. Werden Daten gelesen, tauschen Host und die Bootloader die Rollen. Eine Erläuterung dabei notwendiger steuernder Eingriffe wird später gegeben.

5. Deaktivieren der Bootloader
   Als Abschluss werden die Bootloader für weitere Aktionen freigegeben.

Das ISP kann zwischen Aktivieren und Deaktivieren wiederholte Vorbereitungen benötigen:

- Komplettes Löschen
  Als Vorbereitung genügt die Auswahl des Speicherbereichs, der gelöscht werden soll.

- Schreiben
  Neben dem Speicherbereich ist auch die Adresse anzugeben, an der geschrieben werden soll. Vorgesehen ist ein kontinuierlicher Modus, um den Overhead zu verringern. Damit genügt die Angabe einer Startadresse. Alle empfangenen Daten werden kontinuierlich an die nachfolgenden Adressen geschrieben. Für nicht kontinuierlich aufeinander folgende Daten sind zwischenzeitliche neue Adressangaben notwendig.

- Lesen
  Neben dem Speicherbereich ist die Adresse anzugeben, ab der Daten aus dem Speicher gelesen werden soll. Die Angabe der als letztes zu lesenden Adresse erfolgt zusammen mit dem entsprechendem Befehl.

Wie eingangs erwähnt, ist das Vorbild des entworfenen Protokolls in den Lösungen aus Abschnitt 3.2 zu finden. Wesentlicher Unterschied ist die Verwendung des Bezeichners einer CAN-Nachricht zur Individualisierung. Damit einher geht auch die Verringerung der Nutzlast einer CAN-Nachricht. Ein Datenbyte muss nun für die Kennzeichnung der Aktion verwendet werden. Der verwendete Request-Response-Ansatz erleichtert die Implementierung, muss jedoch als langsam bewertet werden.

4.2. Bootloader

Ausgangspunkt für den Entwurf des Bootloaders war die Abstraktion von der zugrunde liegenden Hardware. So konnten höhere Programmfunktionen unabhängig entworfen werden, zugleich wird die Portierung auf andere Hardware vereinfacht. Wie aus Abschnitt 4.1 ersichtlich, lässt sich das verwendete Protokoll auch mit anderen Schnittstellen verwenden. Der Entwurf musste auch dies berücksichtigen. Zusammengefasst ergab sich die Forderung nach möglichst unabhängigen Komponenten.
Aus der Funktionsweise des ISP wird ersichtlich, dass mit einem lokalen Puffer gearbeitet werden muss. In diesem sind möglicherweise zu lesende Daten zu speichern. Aus Gründen der Performance ist es sinnvoll, auch schreibende Daten zu puffern. Vor diesem Hintergrund bietet sich die Möglichkeit an, alle Komponenten lediglich über Puffer miteinander kommunizieren zu lassen. Die Verbindung zwischen den Komponenten wird über Nachrichten hergestellt, welche über eine zentrale Instanz verteilt werden. Abbildung 2 zeigt das Komponentendiagramm des Entwurfs. Die funktionalen Komponenten des Entwurfs, die Server, kommunizieren untereinander über die mit Kernel bezeichnete Instanz. Dazu signalisieren sie dem Kernel eine ausgehende Nachricht, dieser dann entsprechend weiter reicht. Ein Server kennzeichnet eine Nachricht durch eine Richtung. Entweder die Nachricht soll in der Serverhierarchie nach oben wandern oder nach unten. Bei Unklarheit über die Hierarchie wird die Richtung entsprechend als unbekannt definiert.

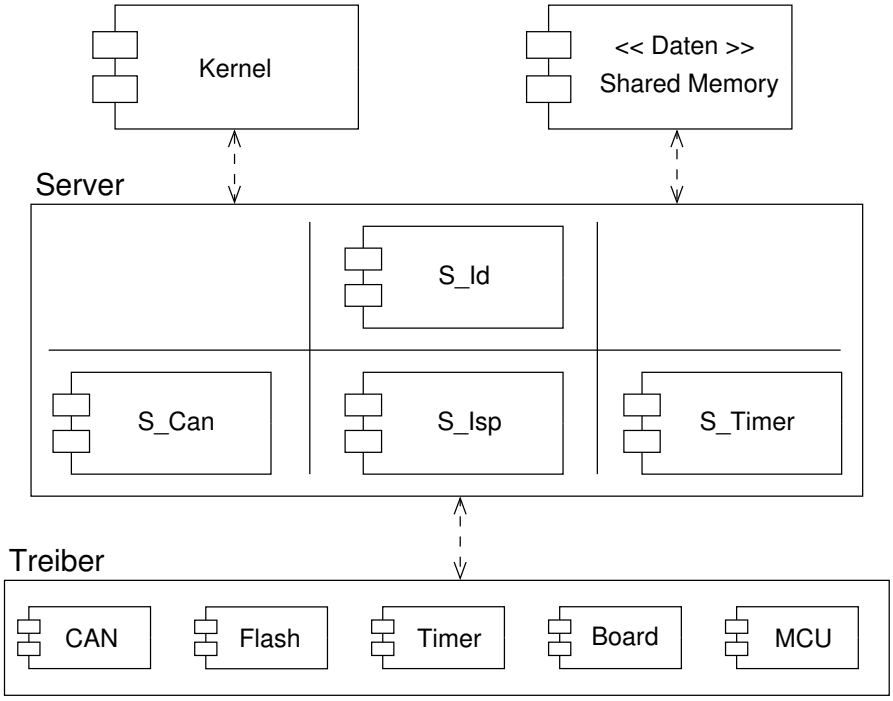

Abbildung 2: Komponentendiagramm des Bootloader

Ist kein Empfänger für ein Signal existent, wird der Sender benachrichtigt. Dadurch wird es möglich, unabhängig von der jeweiligen Konfiguration zu bleiben. Ist beispielsweise kein Debugging gewünscht, wird der zuständige Server aus dem System genommen. Derzeit wird dies lediglich zur Kompilierzeit getätigt. Ein Entfernen auch zur Laufzeit sollte im Bedarfsfall keinen großen Aufwand erfordern. Weiterhin wird die Nutzung von Interrupts möglich. Anstatt an verschiedenen Punkten auf externe Ereignisse zu pollen, braucht mit dem Kernel lediglich eine zentrale Instanz auf nun interne Ereignisse zu pollen. Interne Ereignisse sind die Nachrichten, die von den einzelnen Servern, möglicherweise in Reaktion auf Interrupts, signalisiert werden.
Die Nachteile dieses Entwurfs sind eine verringerte Geschwindigkeit und ein erhöhter Speicherbedarf. Letzteres ist aufgrund der Anforderungen unbedingt während der Implementation zu beachten.

4.3. Programmiersoftware

Aufgrund vorheriger Erfahrungen und der angebotenen Möglichkeiten wurde sich für eine Erweiterung der in Abschnitt 3.3 genannten Programmiersoftware *Avrdude* entschieden. Die Schnittstelle und die Anwendung bleiben daher für bisherige Benutzer gleich. Auf Entwurfseite konnte auf das Handling mit Dateien und das Benutzerinterface verzichtet werden.
Ohne Anspruch auf Vollständigkeit, gibt Abbildung 3 einen Überblick über die Komponenten von *Avrdude*. Ersichtlich wird, dass eine Erweiterung sich auf zwei neue Komponenten beschränken kann. Zunächst ist es notwendig, eine zusätzliche Schnittstelle

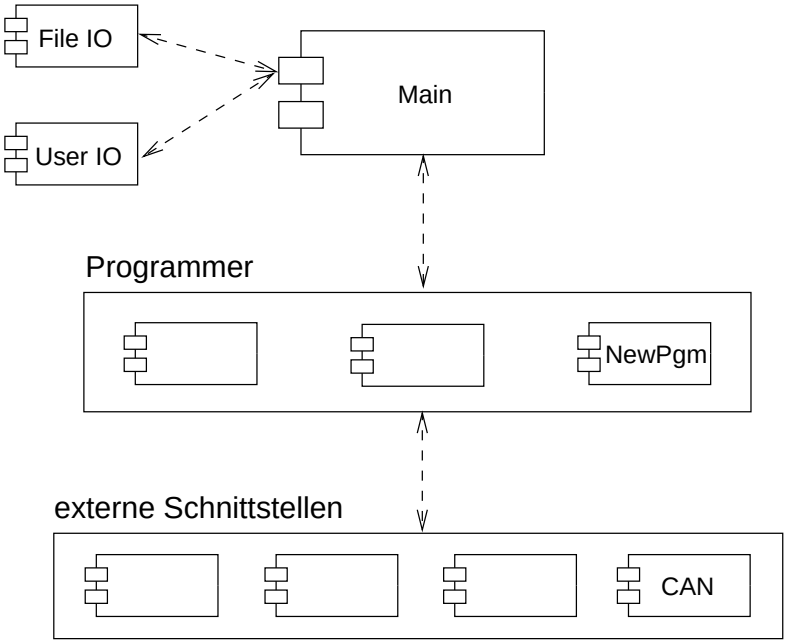

Abbildung 3: Komponentendiagramm von Avrdude

für den CAN-Bus zu entwickeln. Die Funktionalität kann sich dabei an den bereits existierenden Schnittstellen orientieren. Weiterhin muss ein neuer Programmer entwickelt werden. Dieser stellt, auf höherer Ebene, den Adapter zwischen *Avrdude* und einem verwendetem Befehlprotokoll dar. Sämtliche spezifische Funktionalität bleibt in dieser Komponente gekapselt. Daher kann auch die Unterstützung für mehrere Kommunikationspartner in dieser Komponente realisiert werden.

5. Implementierung

Die Umsetzung des Entwurfs aus Abschnitt 4 umfasste den Bootloader sowie das Gegenstück auf Seite des PC. Wesentlicher Teil der jeweiligen Implementation war die Unterstützung des in Abschnitt 4.1 erläuterten Protokolls.

5.1. Bootloader

Zur Implementierung des Bootloaders wurde sich für die Verwendung der in Abschnitt 3.1 genannten Möglichkeiten entschieden. Das erlaubte zwischenzeitliche Funktionstests ohne zusätzlichen Aufwand, wurde doch unter dem gleichen Betriebssystem entwickelt wie in der späteren Anwendung.
Die Wahl der Programmiersprache fiel auf C++. Mit Verwendung dieser Sprache wurde es möglich, für höhere Programmfunktionalität einen objektorientierten Ansatz zu verwenden. Entsprechend konnte die Struktur übersichtlich gehalten werden. Zu beachten waren Nachteile bezüglich des Umfangs des Programmcode, resultierend aus automatisch generiertem Code, beispielsweise für Konstruktoren oder Destruktoren. Mit C++ wurde es gleichzeitig möglich, für die untere Programmfunktionalität reine C-Implementationen zu verwenden. Dies betraf die Schnittstellen zur Hardware. Hier wurde der imperative Ansatz von C bezüglich Einfachheit und Ressourcenbedarf als vorteilhaft eingeschätzt.
In Fortführung des Entwurfs aus Abschnitt 4.2 ergab sich eine aus verschiedenen Klassen bestehende Architektur. Aus Gründen der Übersichtlichkeit werden die entsprechenden Klassendiagramme geteilt. Abbildung 4 stellt die Architektur der Server dar. Wie im

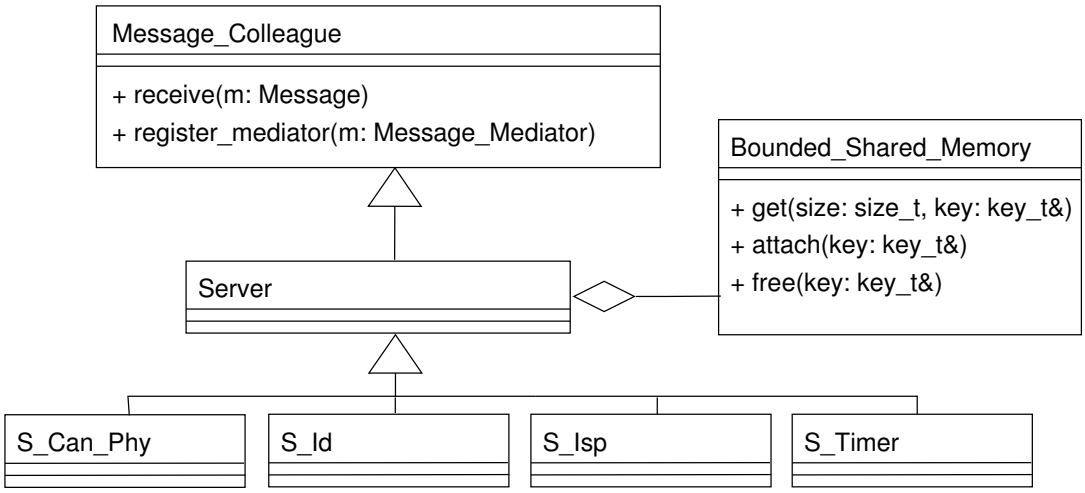

Abbildung 4: Klassendiagramm der Server

Entwurf erläutert, beinhalten diese die eigentliche Funktionalität. Zu sehen ist, dass die Server untereinander minimal gekoppelt sind. Alle verwenden einen gemeinsamen Speicher, von dem sie bei Bedarf Teile reservieren, bearbeiten und wieder freigeben. Inhalt der Teile sind im Normalfall Nachrichten an andere Server, aber auch zu schreibende oder gelesene Daten aus bzw. von Bereichen des Programmspeichers. Das Befehlpro-

tokoll wird nur innerhalb der *receive*-Routine realisiert. Aufgrund der Anzahl der verschiedenen Befehle vollzieht diese Routine im Wesentlichen eine Fallunterscheidung.
Das angewandte Mediator-Pattern setzt sich in den in Abbildung 5 dargestellten Elementen fort. Die Server kommunizieren untereinander durch das Signalisieren von Nachrich-

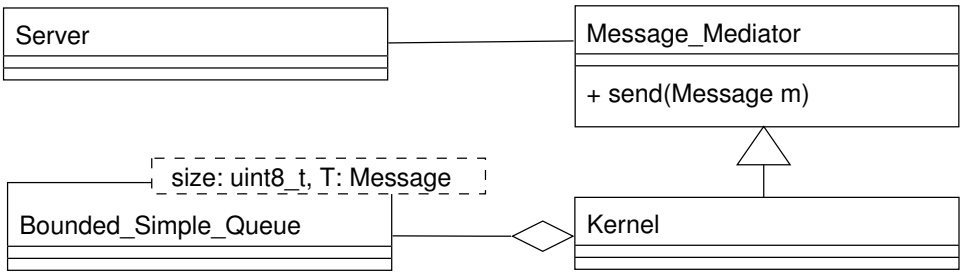

Abbildung 5: Klassendiagramm des Kernel

ten an den Kernel. Wie bereits angedeutet, können einzelne Server in Reaktion auf Interrupts selbstständig aktiv werden. Zu diesem Zweck bieten derartige Server statische Einstiegspunkte für Interrupt Service Routinen. Um in einem solchen Fall mit anderen Servern kommunizieren zu können, müssen die nebenläufig generierten Nachrichten zwischengespeichert werden. Diesem Zweck dient die Queue des Kernels. Deren Abarbeitung nimmt der Kernel innerhalb einer Endlosschleife vor. Liegt eine Nachricht vor, wird der passenden Server gesucht und dessen *receive*-Routine aufgerufen. Nach Beendigung der Routine wird die nächste Nachricht abgearbeitet usw. Dementsprechend erfolgt die Abarbeitung der Nachrichten streng in einem Thread. Nachrichten selbst können wiederum beim Abarbeiten einer Nachricht in die Queue gestellt werden. Dies kann aber auch, wie beschrieben, in Reaktion auf einen Interrupt geschehen.
Die Namen der einzelnen Server geben deren Aufgaben wieder. Der Server *S_Can_Phy* bildet die Schnittstelle zum CAN-Bus. Nachrichten auf dem Bus werden interruptgesteuert an den Kernel signalisiert. Umgekehrt realisiert das Abarbeiten einer Nachricht das Versenden von Daten und die Steuerung der CAN-Schnittstelle. Der Server *S_Id* agiert als Filter für Nachrichten, die vom CAN-Bus stammen. Das ISP wird von Server *S_Isp* durchgeführt. Beinahe das gesamte Befehlprotokoll der Entwicklung ist in diesem Server realisiert. Das Schreiben bzw. Löschen eines Programmspeichers wird ebenfalls interruptgesteuert vorgenommen. Die derzeitige Implementierung verlangt jedoch ein Warten der weiteren Abarbeitung, bis ein ISP-Vorgang abgeschlossen ist. Der Server *S_Timer* dient dem Start einer Applikation nach Ablauf einer vorgegebenen Zeitspanne. Von der Darstellung der in C implementierten Funktionalität wird abgesehen. Diese sind eng mit der Hardware verknüpft. Eine allgemeine Architektur lässt sich hier nicht angeben. Es war jedoch nicht Ziel der Arbeit, für die hardwarenahe Funktionalität eine allgemeine Architektur zu realisieren.

5.2. Programmiersoftware

Aufgrund der Erweiterung von *Avrdude* war die Art und Weise der Implementierung bereits vorgegeben. So musste die Programmiersprache C verwendet werden. Die API von *Avrdude* legte die zu implementierenden Routinen bereits fest.

Zentrales Element der API ist eine mit *PROGRAMMER* bezeichnete Struktur. Diese wird durch Scannen und Parsen eines Config-Files für die spätere Verwendung vorbereitet. Zum Einsatz kommen dafür die unter Linux klassischen Programme *lex* und *bison*. Im Anschluss beinhaltet die *PROGRAMMER*-Struktur - unter anderem - Zeiger auf spezifische Routinen. *Avrdude* geht im eigentlichem Programmlauf nun entsprechend seines Algorithmus' vor. Über die *PROGRAMMER*-Struktur werden spezifische Routinen aufgerufen, wann immer der allgemein gültige Pfad verlassen wird.
Neben den obligatorischen Routinen wurden in der Erweiterung zwei Routinen zum Schreiben und Lesen implementiert. Sämtliche neuen Routinen wurden in der direkt für *Avrdude* sichtbaren Komponente *Shumway* zusammengefasst. Der Name der Komponente ist das Resultat eines Wortspiels. Daran beteiligt sind der Zweck der Erweiterung, das Flashen, und die Namen der Hauptfiguren zweier Fernsehserien, Flash Gordon und Gordon Alf Shumway.
Sämtliche Aufrufe gehen über die zweite entwickelte Komponente, der Schnittstelle zum CAN-Bus. Abbildung 6 gibt einen Einblick in beide Komponenten.

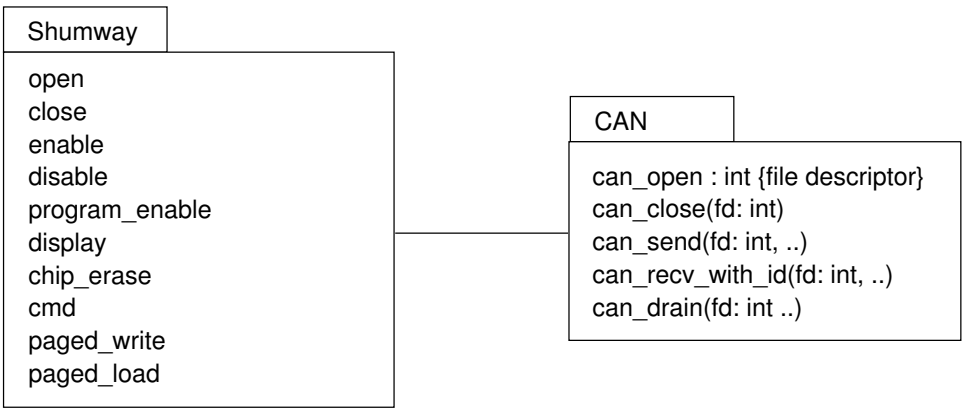

Abbildung 6: Struktur der Erweiterung von Avrdude

Die Implementierung der Schnittstelle zum CAN-Bus versucht, die Trennung vom verwendetem Treiber zumindest vorzubereiten. Gemäß den Anforderungen wurde die *libpcan* für die Implementation verwendet. Abweichend vom unter Unix üblichen Filedescriptor, benutzt diese Bibliothek ein eigenes Konstrukt zur Beschreibung einer Schnittstelle. Im Zusammenspiel mit *Avrdude* ergab sich damit ein Problem. Wie schon erwähnt, ruft *Avrdude* spezifische Routinen entsprechend seinem Ablauf ab. Eine einmal geöffnete externe Schnittstelle muss daher zwischengespeichert werden. Dies geschieht in der *PROGRAMMER*-Struktur, welche den üblichen Filedescriptor beinhaltet. Das Konstrukt der *libpcan*, das sog. Handle, beinhaltet seinerseits einen Filedescriptor, so dass ein Zwischenspeichern möglich ist. Jedoch erwarten sämtliche Routinen der *libpcan* ein Handle als Argument, ein Filedescriptor wird nicht unterstützt. Ein zusätzliches Mapping zwischen beiden Konstrukten ist daher notwendig. Wie aus Abbildung 6 ersichtlich, wurde dies in der für die *libpcan* spezifischen Implementation realisiert. Damit konnte das Interface von *Avrdude* zur CAN-Schnittstelle unabhängig von der Implementation gehalten werden.

6. Anwendung

Vor einer Anwendung der entwickelten Lösung sind mehrere Schritte notwendig. Zunächst müssen die Quellen in ausführbaren Programmcode übersetzt werden. Anschließend müssen die Programme installiert werden. Waren beide Schritte erfolgreich, kann schlussendlich die Anwendung erfolgen.

6.1. Einrichten und Übersetzen

Zum Übersetzen des Bootloaders sollte das beiliegende Makefile verwendet werden. Der Pfad zum Compiler ist ggf. in der Variable *DIRAVR* zu setzen. Zwei Einrichtungsaufgaben sind durchzuführen:

- Einstellen der Baudrate
  Fest einzustellen ist die Baudrate, mit welcher der Bootloader auf dem CAN-Bus kommuniziert. Dazu sind in der Datei „config.h“ im Verzeichnis „config“ die Einstellungen für die entsprechenden *CANBT*-Register zu treffen. Voreingestellt ist eine Baudrate von 250 kbit/s.

- Einstellen des Bezeichners
  Wie in Abschnitt 4.1 erläutert, muss jeder Teilnehmer einen eindeutigen Bezeichner besitzen. Dazu ist die Routine *get_own_id()* aus der Datei „board_drv_ktb_can128.h“, zu finden unter „lib/board“, entsprechend anzupassen. Im Bedarfsfall können eigene Befehle zur Festlegung des Bezeichners eingebracht werden.

Für das Übersetzen von *Avrdude* wird auf die der Software beiliegenden Dokumentation verwiesen. Die getätigte Erweiterung hat keinen Einfluss auf das dort erläuterte Vorgehen. Explizit zu nennende Einrichtungsaufgaben sind nicht bekannt.

6.2. Installieren

Das Installieren des Bootloaders beschränkt sich auf das Übertragen des Programmcode auf den Mikrocontroller. Dazu kann eine beliebige Programmiersoftware, beispielsweise auch *Avrdude*, verwendet werden. Die Prozedur unterscheidet sich im Einzelnen je nach Programmiersoftware und Programmieradapter. Bei Verwendung von *Avrdude* in Verbindung mit einem STK200-kompatiblen Adapter lautet der Aufruf beispielsweise wie folgt:

```
avrdude -p at90can128 -P /dev/parport0 -c stk200 -U flash:w:shumway.hex
```

Dieser Befehl spricht einen AT90CAN128 über einen am ersten Parallelport befindlichen Adapter an.
Abschließend ist durch Setzen der Fuse-Bits sicherzustellen, dass nach einem Reset der Bootloader korrekt gestartet wird. Für den AT90CAN128 bedeutet dies ein Löschen der drei niederwertigsten Bits des Fuse-High-Bytes. Soweit bekannt, kann dies ebenfalls mit gängiger Programmiersoftware erreicht werden. Analog zu obigem Beispiel, lautet der Aufruf von *Avrdude* dazu:

```
avrdude -p at90can128 -P /dev/parport0 -c stk200 -U hfuse:w:0xD8:m
```

Die verwendete Bitmaske 0xD8 entspricht den Standardeinstellungen.
Für die Installation von *Avrdude* wird abermals auf die der Software beiliegenden Dokumentation verwiesen.

6.3. Anwendung

Für den Anwender beschränkt sich die Anwendung auf die Bedienung von *Avrdude*. Aus diesem Grunde ist auch hier die Dokumentation zu diesem Programm hilfreich.
Die Konfiguration von *Avrdude* ist in der Datei „avrdude.conf “ vorzunehmen. Mehrere Einstellungen betreffen die Erweiterung.

- *default_can*
  Dies ist die Festlegung der voreingestellten Schnittstelle zum CAN-Bus. Fehlt beim Aufruf von *Avrdude* die Angabe der Schnittstelle, wird die hier aufgeführte Schnittstelle verwendet. Voreingestellt ist die Schnittstelle */dev/pcan24*, die für die in den Anforderungen genannten PCAN-Dongles zutreffend ist.

- *can_id_host*
  Wie die Bootloader, muss auch *Avrdude* einen eindeutigen Bezeichner für Nachrichten auf dem CAN-Bus verwenden. Mit dieser Einstellung wird der Bezeichner festgelegt. Voreingestellt ist der Wert 0.

- *can_use_ext_id*
  Diese Einstellung ermöglicht die Benutzung von erweiterten Bezeichnern auf dem CAN-Bus. Da der Bootloader dies zwingend fordert, ist eine Anwendung bisher nicht möglich und die Einstellung möglichen Weiterentwicklung vorbehalten. Der Wert ist unbedingt auf der Voreinstellung ’yes’ zu belassen.

- *can_expected_nodes_num*
  Für das Programmieren wurde auf eine individuelle Adressierung verzichtet. Daher bleibt die Anzahl der angesprochenen Mikrocontroller zunächst *Avrdude* überlassen. Dies kann der eigentlichen Absicht des Anwenders widersprechen. Unter Umständen befinden sich irrtümlich Mikrocontroller im Bootloader-Modus, für welche der anlaufende Programmiervorgang nicht gedacht ist. Für diesen Fall kann mit dieser Einstellung die Anzahl der erwarteten Mikrocontroller vorgegeben werden. Stellt *Avrdude* den Kontakt mit mehr oder mit weniger Mikrocontrollern her, wird der Programmiervorgang nicht gestartet. Die Voreinstellung von 0 weist *Avrdude* an, keine Vorgabe zu verwenden. Somit werden alle Mikrocontroller programmiert, zu denen der Kontakt hergestellt werden konnte.

Der Name des von *Avrdude* zu verwendenden Programmieradapters lautet ’pcan’. Insgesamt wird das Programmieren durch folgenden Aufruf von *Avrdude* gestartet:

```
avrdude -p at90can128 -c pcan -U flash:w:<datei>
```

Anstelle des Speicherbereichs Flash kann auch der Speicherbereich EEPROM gewählt werden. Für diesen Fall lautet der Aufruf von *Avrdude*:

```
avrdude -p at90can128 -c pcan -U eeprom:w:<datei>
```

So genannte Verbose-Optionen können *Avrdude* dazu auffordern, mehr Informationen über den Programmlauf zu liefern. Nach der Anzahl der verwendeten Optionen lassen sich vier Stufen der Ausgabe unterschieden.

0 - Keine zusätzliche Ausgabe von Informationen.

1 - Aufgetretene Fehler werden weiter erläutert.

2 - Informationen über den Programmlauf werden ausgegeben.

3 - Ausgabe von Informationen der verwendeten Schnittstelle.

Die Stufen bauen aufeinander auf, eine Stufe beinhaltet somit alle vorhergehenden. In der normalen Anwendung sollte sich jedoch die Verwendung der Verbose-Optionen erübrigen. Eine beispielhafte Ausgabe von *Avrdude* ist in Anhang B aufgeführt.

Wie in 2.4 gefordert, sollte für programmierte Mikrocontroller die Möglichkeit bestehen, ohne Reset ein erneutes Programmieren zu ermöglichen. Zu diesem Zweck muss eine laufende Mikrocontroller-Anwendung an eine vorbereitete Adresse springen. Für eine bequeme Implementierung einer derartigen Funktionalität wurde die Routine 'loader_run()' vorbereitet. Diese befindet sich in der Header-Datei 'loader.h', zu finden im Verzeichnis 'frame' im Quellcode des Bootloader. Eine Anwendung für den Mikrocontroller braucht lediglich diese Header-Datei einzubinden. Nach erfolgreichem Kompilieren und Programmieren kann zur Laufzeit der Anwendung die genannte Routine aufgerufen werden, um ein erneutes Programmieren zu ermöglichen.

6.4. Evaluation

Im Folgendem soll ein Einblick in die Geschwindigkeit des Programmiervorgangs gegeben werden. Für den Erhalt der Angaben wurde jeweils ein Programmiervorgang mit zwei angeschlossenen Mikrocontrollern durchgeführt. Die auf dem CAN-Bus verwendete Baudrate betrug 250kbit/s. Sämtliche Werte sind lediglich Richtwerte und sollen die zu erwartenden Zeiten verdeutlichen.
Tabelle 1 gibt die Zeiten für das Schreiben in den Flash-Speicher der Mikrocontroller an. Die Zeit zum Löschen bleibt konstant, da dies eine selbstständige Aktion der Mikrocontroller ist. Die dafür benötigte Zeit ist durch die Hardware vorgegeben.

Anzahl Daten	Zeit für Löschen	Zeit für Schreiben	Zeit für Verify
2090 Bytes	2,3s	1,50s	1,63s
67626 Bytes	2,3s	44,38s	51,73s
120874 Bytes	2,3s	79,65s	93,60s

Tabelle 1: Zu erwartende Zeiten für den Programmiervorgang des Flash-Speichers

In Tabelle 2 sind die Zeiten aufgelistet, wie sie für das Schreiben in den EEPROM-Speicher zu erwarten sind. Auch hier ist die Zeit zum Löschen von der Hardware bedingt. Wie zu sehen ist, sind Schreibzugriffe auf das EEPROM sehr langsam. Auch

Anzahl Daten	Zeit für Löschen	Zeit für Schreiben	Zeit für Verify
64 Bytes	34,8s	0,61s	0,11s
2128 Bytes	34,8s	19,33s	1,71s
3792 Bytes	34,8s	34,37s	3,04s

Tabelle 2: Zu erwartende Zeiten für den Programmiervorgang des EEPROM-Speichers

ist die Anzahl garantierter Schreib- bzw. Löschzyklen vergleichsweise gering. Die Implementierung berücksichtigt diese Umstände, indem nur geschrieben wird, wenn dies erforderlich ist. Sind zu schreibende Werte bereits im EEPROM vorhanden, wird auf diesem Wege Zeit gespart und die Lebensdauer erhöht. Die angegebenen Werte sind als Maximalwerte zu betrachten, da sie auf einem vorher gelöschten EEPROM beruhen.
Avrdude versucht, vor dem Schreiben in den Flash-Speicher alle Speicherbereiche eines Mikrocontrollers zu löschen. Im Rahmen dieses sog. Chip-Erase betrifft das auch den EEPROM-Speicher. Dazu sind jedoch im schlechtesten Fall mehr als 34 Sekunden notwendig. Einem Anwender, der lediglich das Flash beschreiben will, ist diese Wartezeit nicht zuzumuten. Daher wurde der Bereich EEPROM vom Chip-Erase ausgenommen. Da bedeutet auch, dass der EEPROM-Speicher nicht gelöscht werden kann. Selbstverständlich können aber neue Daten abgelegt werden.
Im Vergleich zur herkömmlichen Programmierung sind die Zeiten für den EEPROM-Speicher annähernd gleich. Lediglich im Verify werden wenige Sekunden zusätzlich benötigt. Für den Bereich des Flash-Speichers halbiert sich die Geschwindigkeit nahezu. Die Gründe dafür konnten bis zum Abschluss der Arbeit nicht geklärt werden. Das verwendete Request-Response-Verfahren ist sicherlich kein schnelles Verfahren. Auf jeweils sieben Bytes Daten muss eine Bestätigung versandt und abgewartet werden. Der CAN-Bus sollte aufgrund der verwendeten Baudrate keinen Engpass darstellen. Die Häufigkeit der auf den Mikrocontrollern laufenden ISP-Prozeduren hat ebenfalls nur geringen Einfluss. Versuche mit unterschiedlichen Puffergrößen sind zu diesem Zwecke erfolgt. Auf Seite von *Avrdude* sind keine bedeutenden Verzögerungen bekannt. Ein Wechsel der verwendeten Bibliothek für die CAN-Schnittstelle ist aufgrund der Anforderungen nicht möglich. Als bedeutende Ursache vermutet wird eine lange Verarbeitungszeit auf dem Mikrocontroller, bedingt durch das vielfache Message-Handling. Ein direktes Mapping zwischen den Servern, welches möglicherweise einen schnelleren Ablauf bewirken würde, konnte mangels Speicherplatz nicht eingesetzt werden. Denkbar ist weiterhin, dass das häufige Warten auf den Abschluss eines ISP-Vorgangs Zeit kostet. Ein erster Test konnte diese Vermutung jedoch nicht bestätigen.

7. Zusammenfassung und Ausblick

Mit der entwickelten Lösung ist es möglich, gleichzeitig mehrere Mikrocontroller der AT90CAN-Baureihe über deren CAN-Schnittstelle zu programmieren. Dem Benutzer wird mit der Erweiterung des Programms *Avrdude* ein entsprechendes Werkzeug zur Anwendung gegeben. Die gesamte Entwicklung erfolgte unter Verwendung von offen gelegter Software unter dem Betriebssystem Linux. Da Avrdude den Quasi-Standard für Programmiersoftware unter Linux darstellt, wird der Aufwand zur Anwendung als minimal angenommen.
Nicht realisiert wurden Möglichkeiten zum Programmieren der Fuse-Bytes. Ebenso fehlt ein eigener Update-Mechanismus. Die Entwicklung der genannten Punkte scheiterte am verfügbaren Speicherplatz. Allerdings kann aufgrund der Architektur der Lösung angenommen werden, dass entsprechende Modifikationen einfach durchführbar sind. Ob eine Variante jemals alle Bereiche programmieren kann, oder aber verschiedene spezialisierte Varianten existieren, wird nicht zuletzt durch den Bedarf entschieden. Gleiches trifft auch für Verbesserungen hinsichtlich der Performance zu.

Anhang A. Befehlprotokoll

A.1. Öffnen

Aktion	Daten	Beschreibung
SELECT_OPEN	-	Öffnet Empfänger für weitere Nachrichten von diesem Sender

Tabelle 3: Anfrage des Hosts nach Öffnen

Aktion	Daten	Beschreibung
SELECT_OPEN	OK	Empfänger für weitere Nachrichten von diesem Sender bereit
SELECT_OPEN	ERROR	Empfänger konnte ungestörten weiteren Empfang nicht sicherstellen

Tabelle 4: Antwort des Bootloaders auf Öffnen

A.2. Schließen

Aktion	Daten	Beschreibung
SELECT_CLOSE	-	Beenden des Empfangs weiterer Nachrichten dieses Senders

Tabelle 5: Anfrage des Hosts nach Schließen

Aktion	Daten	Beschreibung
SELECT_CLOSE	OK	Empfänger geschlossen

Tabelle 6: Antwort des Bootloaders auf Schließen

A.3. Speicherwahl

Aktion	Daten	Beschreibung
MEM_SELECT	MEM	Auswahl einer Speichers für weitere Aktionen

Tabelle 7: Anfrage des Hosts nach Wahl eines Speichers

Aktion	Daten	Beschreibung
MEM_SELECT	OK	Speicherwahl erfolgreich
MEM_SELECT	ERROR	Speicherwahl ungültig

Tabelle 8: Antwort des Bootloaders auf Wahl des Speichers

A.4. Adresswahl

Aktion	Daten	Beschreibung
ADDR	Adresse	Auswahl einer Adresse für weitere Aktionen

Tabelle 9: Anfrage des Hosts nach Wahl der Adresse

Aktion	Daten	Beschreibung
ADDR	OK	Adresswahl erfolgreich
ADDR	ERROR	Adresswahl ungültig

Tabelle 10: Antwort des Bootloaders auf Wahl der Adresse

A.5. Speicher komplett löschen

Aktion	Daten	Beschreibung
FULL_ERASE	-	Löschen des gesamten Speicherbereichs

Tabelle 11: Anfrage des Hosts nach komplettem Löschen

Aktion	Daten	Beschreibung
FULL_ERASE	OK	Löschen in Durchführung
FULL_ERASE	ERROR	Löschen kann nicht durchgeführt werden

Tabelle 12: Antwort des Bootloaders auf komplettes Löschen

A.6. Schreiben

Aktion	Daten	Beschreibung
WRITE	Programmdaten	Schreiben von Daten

Tabelle 13: Anfrage des Hosts nach Schreiben

Aktion	Daten	Beschreibung
WRITE	OK	Daten akzeptiert
WRITE	ERROR	Fehler, Daten nicht akzeptiert

Tabelle 14: Antwort des Bootloaders auf Schreiben

A.7. Lesen

Aktion	Daten	Beschreibung
READ	letzte Adresse	Lesen von Daten

Tabelle 15: Anfrage des Hosts nach Lesen

Aktion	Daten	Beschreibung
READ	gelesene Daten	Übermittlung gelesener Daten
WRITE	OK	Alle Daten bis zur letzten Adresse gelesen
READ	ERROR	Fehler beim Lesen der Daten

Tabelle 16: Antwort des Bootloaders auf Lesen

Anhang B. Programmiervorgang mit Avrdude

Für den Programmiervorgang verwendet wurden zwei AT90CAN128. Beide befanden sich bei Beginn im Bootloader-Modus. Abweichend von der üblichen Vorgehensweise, wurde *Avrdude* über die angegebene Datei „avrdude.conf“ konfiguriert

```
[diederic@eoslab-09 avrdude-5.1-can]$ ./avrdude -C avrdude.conf -p at90can128 -c
pcan -U flash:w:../../lab_prak/scratch/CAN/can.hex

avrdude: Connected to 2 nodes via CAN.
avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x9d9d9d
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: current erase-rewrite cycle count is -1886417009 (if being tracked)
avrdude: erasing chip
avrdude: reading input file "../../lab_prak/scratch/CAN/can.hex"
avrdude: input file ../../lab_prak/scratch/CAN/can.hex auto detected as Intel Hex
avrdude: writing flash (2090 bytes):

Writing | ################################################## | 100% 1.50s



avrdude: 2090 bytes of flash written
avrdude: verifying flash memory against ../../lab_prak/scratch/CAN/can.hex:
avrdude: load data flash data from input file ../../lab_prak/scratch/CAN/can.hex:
avrdude: input file ../../lab_prak/scratch/CAN/can.hex auto detected as Intel Hex
avrdude: input file ../../lab_prak/scratch/CAN/can.hex contains 2090 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 1.63s



avrdude: verifying ...
avrdude: 2090 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done.  Thank you.

[diederic@eoslab-09 avrdude-5.1-can]$
```

Aufgrund fehlender Unterstützung seitens der Entwicklung gibt *Avrdude* zwei fehlerhafte Informationen aus.

- Es wird keine Gerätesignatur ausgelesen. Die angezeigte Signatur ist lediglich der zufällige Inhalt einer Speicherzelle. Das Auslesen einer Signatur ist von der Entwicklung nicht vorgesehen. Dementsprechend ist eine Routine zum Auslesen einer Signatur nicht in *Avrdude* implementiert. Eine Implementation muss aber nicht erfolgen, die Routine ist als optional gekennzeichnet.
- Es wird kein Zählen der Schreib- und Löschzyklen durchgeführt. *Avrdude* ermöglicht es, vier Bytes im EEPROM abzulegen, der dann als Zähler der getätigten Zyklen dient. Das Auslesen dieser Information basiert wiederum auf einer als optional gekennzeichneten Routine. Obwohl diese nicht implementiert ist, verwendet *Avrdude* einen vermeintlich gelesenen Wert und gibt diesen aus.

Literaturverzeichnis

[1] ATMEL CORPORATION: *8-bit AVR Microcontroller with 32K/64K/128K Bytes of ISP Flash and CAN Controller*. Rev. 7679A-CAN-10/06, http://www.atmel.com/dyn/resources/prod_documents/doc7679.pdf

[2] ATMEL CORPORATION: *AVR914: CAN & UART based Bootloader for AT90CAN32, AT90CAN64, & AT90CAN128*. Rev. 7592B-AVR-01/06, http://www.atmel.com/dyn/resources/prod_documents/doc7592.pdf

[3] ATMEL CORPORATION: *"Slim" CAN Boot Loader*. Rev. 1.0.0, http://www.atmel.com/dyn/resources/prod_documents/at90can_lib_3-1.zip

[4] CAN IN AUTOMATION (CIA): *CAN physical layer*. Website: http://www.can-cia.org/can/physical-layer/#media, 2006. – revision 2004-05-26

[5] ETSCHBERGER, K. : *CAN Controller-Area-Network*. 1. Auflage. Carl Hanser Verlag München Wien, 1994

[6] FREE SOFTWARE FOUNDATION: *AVR C Runtime Library*. Website: http://www.nongnu.org/avr-libc/, 2006. – revision 2006/10/09 21:03:34

[7] FREE SOFTWARE FOUNDATION: *AVR Downloader/UploaDEr*. Website: http://www.nongnu.org/avrdude/, 2006. – revision Fri Sep 23 23:21:06 MET DST 2005

[8] FREE SOFTWARE FOUNDATION: *GCC, the GNU Compiler Collection*. Website: http://gcc.gnu.org/, 2006. – revision 2006-09-21

[9] IAR SYSTEMS: *IAR Embedded Workbench–C/C++ compiler and debugger tools*. Website: http://www.iar.com, 2006. – revision 20:00, October 20th