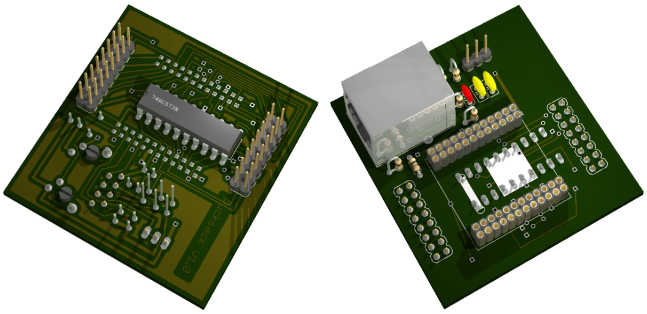
Studienarbeit
Socketbibliothek für einen IIM7000

David Schwerbel
emerald@uni-koblenz.de

Betreuer: Dr. Merten Joost

23. November 2006



Zusammenfassung

Dieses Dokument beschäftigt sich mit der Anbindung des w3100a-TCP/IP–Chips an einen AVR ATmega162 Mikrocontroller. Die Ansteuerung des Netzwerkchips erfolgt, um eine gute Adaptierbarkeit zu gewährleisten, durch eine C–Bibliothek, die in ihrer Funktionsweise der C–Socketbibliothek ähnelt.

Inhaltsverzeichnis

1 Intention

In der heutigen Welt werden immer mehr Geräte des alltäglichen Lebens miteinander vernetzt. Um diese Vernetzung effektiv zu nutzen ist es notwendig, möglichst viele Geräte in diese Netze zu integrieren. Die verbreitetste Netzstruktur ist hierbei der Ethernet-Standard auf den die TCP/IP [1] Netze aufsetzen. Hierüber lassen sich einfach Geräte über große Entfernungen ansprechen und steuern. Oft werden hierfür PCs verwendet die jedoch für einfachere Regelungsaufgaben unwirtschaftlich sind da sie sowohl in der Anschaffung als auch im Unterhalt kostspielig sind. Für Steuerungs- und Regelungsaufgaben sind Mikrocontroller prädestiniert, jedoch können Mikrocontroller in der Regel nicht mit dem Netzwerk verbunden werden, da ihnen die entsprechende Schnittstelle fehlt.

2 Grundlagen

2.1 Mikrocontroller

Ein Mikrocontroller vereint alle Grundbestandteile eines Computers auf einem Chip. Hierzu zählen CPU, RAM und I/O-Komponenten. Sie wurden in der Vergangenheit meist in der Industrie eingesetzt, da sie kostengünstiger, robuster und sparsamer als Computer sind. Seit einigen Jahren findet man aber auch in Privathaushalten immer mehr Mikrocontroller, zum Beispiel in Waschmaschinen, Videorecorder, CD-Playern und vielen anderen Geräten.

2.2 SRAM

Um den verfügbaren Arbeitsspeicher eines Mikrocontrollers zu erweitern, und um andere Geräte wie zum Beispiel LCD-Displays anzusteuern wurde das SRAM bzw. XMEM Interface entwickelt. Dieses Interface wird auch verwendet um den ATmega162 an den IIM7000 Netzwerkchip anzubinden. Das Interface benötigt 16 Adressleitungen um Daten anzufordern und acht Datenleitungen um diese zu übertragen. Zusätzlich werden drei Kontrollleitungen benötigt.

2.3 Ethernet & TCP/IP

Der Ethernet-Standard implementiert die untersten beiden Schichten des OSI-Modells. Er definiert also das Übertragungsmedium, die Art und Weise wie Bits übertragen werden und wie die Bits zu Frames gruppiert werden.
TCP/IP setzt sich aus den Standards TCP und IP zusammen. Während IP in der Vermittlungsschicht des OSI-Modells angesiedelt ist, und verbindungslos arbeitet, wird durch TCP welches verbindungsorientiert arbeitet spezifiziert wie Verbindungen zwischen Geräten aufgebaut werden.

[1] Transmission Control Protocol / Internet Protocol

3 Hardware

Die Firma Atmel bietet eine breite Produktpalette an Mikrocontrollern, deren gute Dokumentation und Verfügbarkeit sie auch für Privatpersonen interessant macht. Aus diesen Gründen, wegen der guten Entwicklungswerkzeuge und der SRAM–Schnittstelle wurde für dieses Projekt ein ATmega162 gewählt.
Als Interface zwischen ATmega162 und Ethernet wird ein IIM7000–Modul verwendet. Dieses besteht wiederum aus einem w3100a–Chip, in den ein kompletter TCP/IP–Stack integriert ist, und einem Realtek–Netzwerkchip. Als Mikrocontroller wird ein ATmega162 verwendet, da dieser über das nötige SRAM Interface verfügt und in der handlichen DIL–Bauform verfügbar ist. Das IIM7000–Modul unterstützt auch eine Kommunikation über den I^2C–Bus. Von der Verwendung des I^2C–Busses wurde jedoch, wegen seiner geringeren Geschwindigkeit, abgesehen.

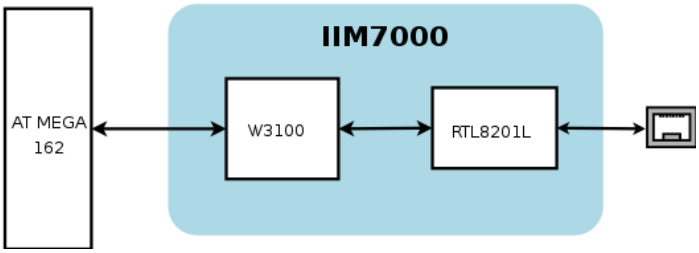


Abbildung 1: Aufbau

Die Verwendung eines TCP/IP–Chips wie des IIM7000 hat verschiedene Vorteile gegenüber der direkten Ansteuerung des Netzwerkchips. Es entfällt das manuelle Generieren der CRC–Prüfsummen, des NLP [2] und der für eine TCP–Verbindung notwendigen Kontrollpakete.
Die Berechnung der Ethernet–CRC–Prüfsumme, der TCP–CRC–Prüfsumme und das Senden von Datenempfangsbestätigungen (Acknowledgments) ist für einen Mikrocontroller aufwändig, benötigt viel Rechenzeit und sind zeitkritisch. Ohne die Verwendung eines TCP/IP–Chips bleibt daher wenig Rechenleistung und Arbeitsspeicher für die Ausführung aufwendiger Applikationen.

3.1 ATmega162

Bei dem verwendeten ATmega162 handelt es sich um einen 8bit RISC[3] Mikrocontroller mit 16KB Flashspeicher und 1KB RAM. Er kann mit einer Taktrate von bis zu 16Mhz betrieben werden und die meisten Operationen in einem Takt ausführen.

[2] der Normal Link Pulse ist ein spezielles Paket, welches in periodischen Abständen von Ethernet-Geräten gesendet wird um anderen Endgeräten mitzuteilen welche Ethernet-Merkmale sie unterstützen (z.B. 100 bzw. 10Mbit)
[3] Reduced Instruction Set Computing

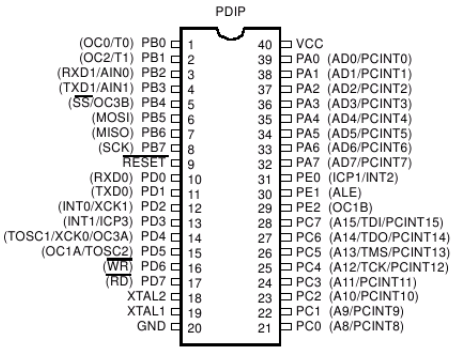


Abbildung 2: Pinbelegung des ATmega162

Durch die Verbreitung der ATmega-Serie entstand eine breite Palette an Werkzeugen und Bibliotheken die die Entwicklung vereinfachen. Hierzu zählen die avr-libc, eine schlanke Variante der GnuLibC, und Programme zum Simulieren und Programmieren der Mikrocontroller.

3.2 IIM7000

Wie bereits erwähnt, besteht das IIM7000–Modul aus einem w3100a–TCP/IP–Chip der Firma WIZnet [WIZ] und einem RTL8201BL–Realtek–Netzwerkchip. Das Modul verfügt über einen separaten 25MHz Quarz und ist damit unabhängig vom Takt des Mikrocontrollers.

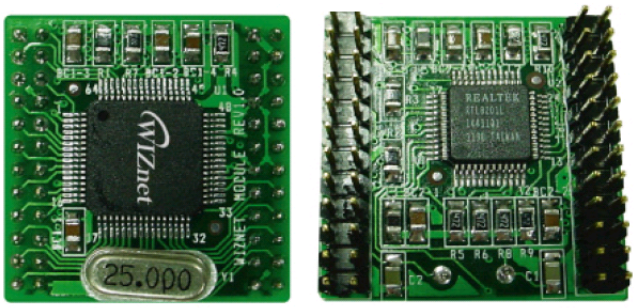


Abbildung 3: Photo eines IIM7000

Die Anbindung des Moduls geschieht über ein SRAM–Interface mit acht Daten- und 15 Adressleitungen, einer Chip–Select–Leitung die bei nur einem SRAM–Gerät auf Masse gelegt wird, einer Write– und einer Read–Leitung über die dem Modul mitgeteilt wird, ob gerade Daten in den Speicher des Moduls geschrieben werden, oder aus ihm gelesen werden. Auf Seite des ATmega162 wurde ein D–FlipFlop verwendet um einen der Ports auf dem die Adresse ausgegeben wird, gleichzeitig als Datenport zu verwenden.

Um das SRAM–Interface zu nutzen muss dieses zunächst im Mikrocontroller konfiguriert und aktiviert werden. Dies geschieht über die Register EMCUCR[4] und MCUCR[5].

```
EMCUCR  |= (1<<SRW11)|(1<<SRW10); // configure SRAM Timings
MCUCR   |= (1<<SRE);    // enable SRAM Interface
```

Zusätzlich verfügt das Modul noch über eine Interrupt Leitung die dem Mikrocontroller mitteilt, dass neue Daten empfangen wurden.
Die Verbindung zur RJ45 [Wikc] Buchse wird über vier Datenleitungen (TPRX+, TPRX-, TPTX+, TPTX-) und vier Leitungen für die in der Buchse enthaltenen LEDs hergestellt [Inc]. Diese LEDs signalisieren, ob Aktivität auf dem Port herrscht, ob es sich um eine 100Mbit oder 10Mbit Verbindung handelt und ob es zu Kollisionen kommt.

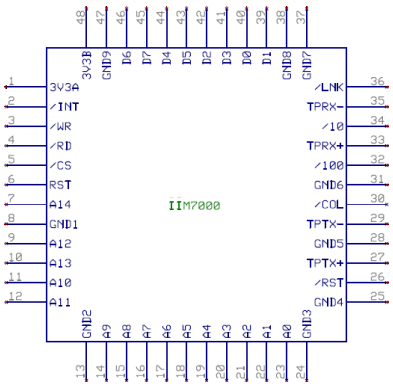


Abbildung 4: Blockbild eines IIM7000

[4]Extended MCU Control Register
[5]MCU Control Register

3.3 Platine

Um die Komplexität des Layouts möglichst gering zu halten wurden die Bauteile auf 2 Platinen aufgeteilt. Eine Basisplatine die den ATmega162, Quarz und die Stromversorgung enthält und eine aufsteckbare Platine die den IIM7000, den D–FlipFlop und die RJ45–Buchse enthält.

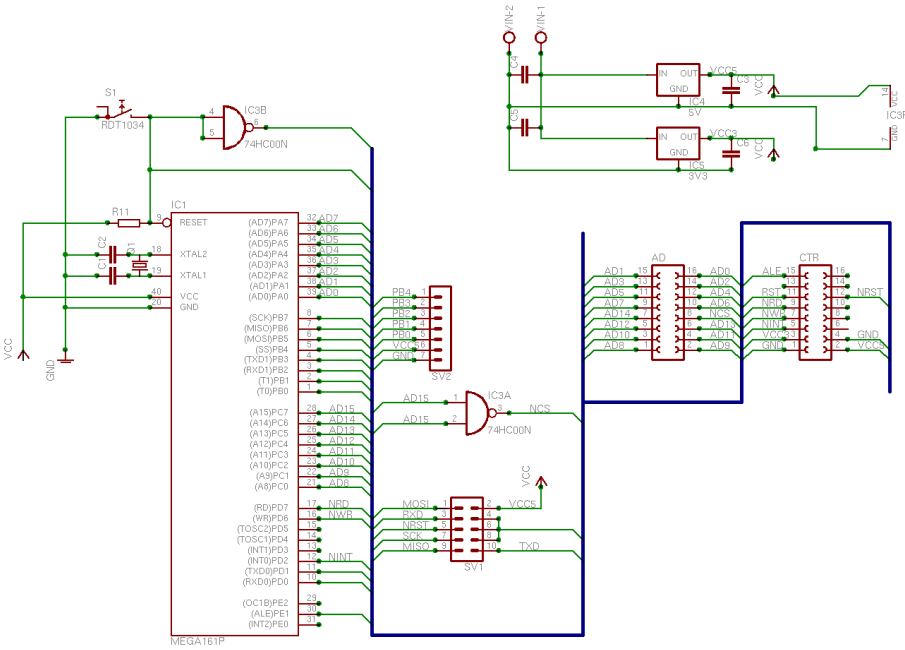


Abbildung 5: Basisplatine

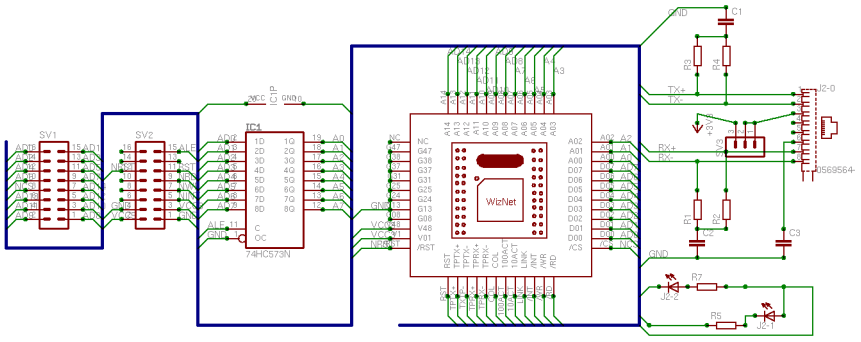


Abbildung 6: Aufsteckplatine

4 Programmierschnittstelle

Ziel dieser Studienarbeit war es, eine der POSIX Socket–Schnittstelle ähnliche Bibliothek, für den Zugriff auf IIM7000–Chip, zu entwickeln. Die Socket Schnittstelle [The] ist Teil des POSIX.1 Standards, der auch als IEEE 1003.1 bekannt ist [Wikb].
Für die Verwendung auf einem Mikrocontroller sind jedoch nicht alle Funktionen notwendig. Die hier implementierten TCP–Funktionen[Wikd] lauten:

Listing 1: Atmel TCP Functions

```
char socket(SOCKET s, u_char protocol, u_int port, u_char flag);
char connect(SOCKET s, u_char* destip, u_int port);
void NBlisten(SOCKET s);
int send(SOCKET s, const u_char* buf, u_int len);
int recv(SOCKET s, const u_char* buf, u_int len);
void close(SOCKET s);
u_int select(SOCKET s, u_char func);
```

Für das Versenden und Empfangen von UDP–Paketen[Wike] wurden folgende Funktionen implementiert:

Listing 2: Atmel UDP Functions

```
u_int sendto(SOCKET s, const u_char*, u_int, u_char*, u_int);
u_int recvfrom(SOCKET s, const u_char*, u_int, u_char*, u_int*);
```

Als Codebasis für die Bibliothek wurde eine Bibliothek von der Firma WIZnet, welche auch den w3100a entwickelt hat genutzt. Leider wurde die Bibliothek mit dem "KEIL C Compiler for AVR"[WIZ] entwickelt, und nicht mit dem frei verfügbaren GCC. Außerdem wurde die Bibliothek für den Atmel AT90s8515 geschrieben, welcher wie der w3100a das "big endian"–Datenformat nutzt. Der nun verwendete Atmel ATmega162 verwendet aber das "little endian"–Datenformat, somit müssen alle Adressen, die dem Modul übergeben werden, vorher konvertiert werden.
Zum besseren Verständnis hier ein kleines Beispiel: [Wika]

Hex-Darstellung	0xA4B3C2D1			
Speicheradresse	50	51	52	53
Speicheroffset (Nummer)	0	1	2	3
Little Endian	D1	C2	B3	A4
Big Endian	A4	B3	C2	D1

Abbildung 7: Big Endian versus Little Endian

4.1 Chip-Initialisierung

Bevor der w3100a–Chip genutzt werden kann, müssen wichtige Parameter wie z.b. die IP–Adresse an den Chip übergeben werden. Zur besseren Übersichtlichkeit und Wartbarkeit kann man diese Parameter zunächst in Variablen speichern und dann an den Chip übertragen. Hier eine kleine Beispielinitialisierung:

Listing 3: Atmel Chip Init

```
#include "type.h"
#include "socket.h"

u_char mask[]={255,255,255,0};
u_char ip[]={192,168,0,2};
% u_int port = 23;
u_char gateway[]={192,168,0,1};
u_char mac[]={0x00,0x11,0xf1,0x1e,0x21,0xa1};
SOCKET s;

int main(int argc, char* argv[])
{
initW3100A();

setgateway(gateway);

setsubmask(mask);

setMACAddr(mac);

setIP(ip);

sysinit(0x55,0x55); // Zuweisung der Sende- und Empfangspuffer

/** CODE **/
}
```

Der w3100a kann bis zu vier Kanäle verwalten, die entweder im TCP, UDP, IP–RAW oder MAC–RAW Modus operieren. Der Chip verfügt insgesamt über 8KB Sendepuffer[6] und 8KB Empfangspuffer[7]. Der Puffer kann relativ frei auf die bis zu vier Kanäle aufgeteilt werden. Dies geschieht über eine Bitmaske die bei der Initialisierung an die Socketbibliothek übergeben wird. Im obigen Beispiel wurde sowohl dem Sende-, als auch dem Empfangspuffer für jeden Socket 2KB Speicher zugewiesen.
Ein paar Beispiele sind hier zu sehen:

Hex-Wert	Bitmaske	Kanal #3	Kanal #2	Kanal #1	Kanal #0
0x00	00000000	1KB	1KB	1KB	1KB
0x03	00000011	unbenutzbar	unbenutzbar	unbenutzbar	8KB
0x0A	00001010	unbenutzbar	unbenutzbar	4KB	4KB
0x55	01010101	2KB	2KB	2KB	2KB

Tabelle 1: Beispiel zur Speicheraufteilung der Puffer

Siehe hierzu 5.2 Speicherverwaltung Seite 20.

[6] im folgenden auch TX Buffer genannt
[7] im folgenden auch RX Buffer genannt

4.2 Socket-Initialisierung

Nachdem der w3100a initialisiert wurde, können nun Sockets erstellt werden. Diese können für den TCP–, UDP– oder IP–RAW–Modus konfiguriert werden, und entweder als "passive open"–Socket[8] auf eingehende Verbindungen warten oder aktiv eine Verbindung aufbauen. Die Wahl des Übertragungsmodus erfolgt über die `setIPprotocol`–Funktion.

```
void setIPprotocol(SOCKET s, u_char IPPROTOCOL);
```

Die Variable `s` vom Typ `SOCKET` spezifiziert, welcher Socket konfiguriert wird. Intern wird entspricht der Typ einem u_int und ist gleich der Nummer des Kanals, also wäre `s=0` der erste Kanal. `IPPROTOCOL` definiert den Wert, der bei gesendeten Paketen im `protocol` Feld des IP–Pakets eingetragen wird. Der Wert muss konform zu RFC790 [Pos] gewählt werden. Folgende Werte sind bereits vordefiniert:

Listing 4: Atmel Socket Protocol

```
#define IPPROTO_ICMP    1       // Internet Control Message Protocol
#define IPPROTO_TCP     6       // Transmission Control Protocol
#define IPPROTO_UDP     17      // User Datagram Protocol
#define IPPROTO_RAW     255     // raw IP
```

Nun kann der Socket mit Hilfe der `socket`–Funktion geöffnet werden:

```
int socket(SOCKET s, u_int SOCK_TYPE, u_int port,u_char 0);
```

Über den zweiten Parameter muss hierbei der Funktion ebenfalls der Protokolltyp übergeben werde, welcher wie folgt gewählt werden kann:

Listing 5: Atmel Socket Type

```
#define SOCK_STREAM     0x01    // TCP
#define SOCK_DGRAM      0x02    // UDP
#define SOCK_IPL_RAW    0x03    // IP RAW
#define SOCK_MACL_RAW   0x04    // MAC RAW
```

Der $\texttt{SOCK}_{\texttt{T}}\texttt{YPE}$ entscheidet darüber welche Art von Socket der w3100a öffnet.

[8] als Server beziehungsweise im "listening"–Modus

4.3 TCP

4.3.1 TCP–Server

Um Dienste über einen TCP–Server anzubieten muss sich der entsprechende Socket im Status *listen* befinden.

```
NBlisten(SOCKET s);
```

Die entsprechende Funktion arbeitet non–blocking, das heißt das Hauptprogramm läuft weiter und muss in regelmäßigen Abständen nachfragen ob eine Verbindung eingegangen ist. Hierzu nutzt man die `select`–Funktion.

```
u_int select(SOCKET s, u_char function);
```

Die `select`–Funktion ist ein wichtiges Hilfsmittel zur Überwachung der TCP Verbindungen. Sie liefert je nach Wahl des Parameters `function`,

- die Größe der Daten im Empfangspuffer, mit dem Parameter `SEL_RECV`
- den freien Speicher im Sendepuffer, mit dem Parameter `SEL_SEND`
- den Status der Verbindung, mit dem Parameter `SEL_CONTROL`.

  Beim Aufruf der `select`–Funktion mit dem Parameter `SEL_CONTROL` liefert sie einen der folgenden Rückgabewerte:

Listing 6: Atmel Socket Status Codes

```
#define SOCK_CLOSE        0x00   // connection closed
#define SOCK_ARP          0x01   // socket in ARP mode
#define SOCK_LISTEN       0x02   // waiting for incomming TCP
connection
#define SOCK_SYNSENT      0x03   // TCP connection setup
#define SOCK_SYNSENT_ACK  0x04   // TCP connection setup
#define SOCK_SYNRECV      0x05   // TCP connection setup
#define SOCK_ESTABLISHED  0x06   // TCP connection established
#define SOCK_CLOSE_WAIT   0x07   // TCP connection teardown
#define SOCK_LAST_ACK     0x08   // TCP connection teardown
#define SOCK_FIN_WAIT1    0x09   // TCP connection teardown
#define SOCK_FIN_WAIT2    0x0A   // TCP connection teardown
#define SOCK_CLOSING      0x0B   // TCP connection teardown
#define SOCK_TIME_WAIT    0x0C   // TCP connection teardown
#define SOCK_RESET        0x0D   // TCP connection teardown
#define SOCK_INIT         0x0E   // socket initialization
#define SOCK_UDP          0x0F   // socket in UDP mode
#define SOCK_RAW          0x10   // socket in IP RAW mode
```

Folgendes Anwendungsbeispiel der `select`–Funktion wartet auf eine eingehende TCP Verbindung:

```
while !((select(socknum,SEL_CONTROL) == SOCK_ESTABLISHED))
       { wait_1us(1); }
```

Durch Kombination mit einem weiteren Aufruf der `select`–Funktion mit Übergabe des Parameters $\texttt{SEL}_{\texttt{R}}\texttt{ECV}$ lässt sich eine Warteschleife programmieren die wartet bis eine Verbindung aufgebaut wurde und Daten empfangen wurden.

```
while !((select(socknum,SEL_CONTROL) == SOCK_ESTABLISHED) && (select(
socknum, SEL_RECV)>1))
        { wait_1us(1); }
```

Um die empfangenen Daten auszulesen benötigt man die `recv`–Funktion:

```
int recv(SOCKET s, const u_char* buf, u_int len);
```

Sie kopiert eine bestimmte Menge an Daten vom Empfangspuffer an die Speicherstelle `buf`. Die Menge der Daten wird über den dritten Parameter `len` übergeben, und über den zweiten Parameter ein Pointer auf die Speicherstelle. Der Rückgabewert gibt die Menge der kopierten Daten in Byte an, bzw. -1 falls die Funktion nicht erfolgreich ausgeführt werden konnte. Es gilt zu beachten, dass nicht mehr Daten kopiert werden als im Empfangspuffer sind und nicht mehr als in den Speicherbereich passen.

Die Kommunikation in Gegenrichtung geschieht über die `send`–Funktion.

```
int send(SOCKET s, const u_char* buf, u_int len)
```

Sie funktioniert analog zur `recv`–Funktion. Sie erwartet als Parameter den Socket, die Speicherstelle an der die zu versendenden Daten liegen und die Menge der Daten. Sie liefert die Anzahl der gesendeten Byte, bzw. -1 als Fehlercode zurück.

Als kleines Beispiel dient hier ein Echo-Server:

Listing 7: Atmel TCP Echo Server

```
1  #define mydatalen 128
2
3  int main(int argc, char* argv[]){
4
5   if ((mydata=malloc(mydatalen))==NULL)
6    puts("malloc -1\n"),exit(0);
7   // allocate memory for the buffer
8
9   /** insert socket initialization here **/
10
11  while (1)
12  {
13   if ((select(socket,SEL_CONTROL) == SOCK_ESTABLISHED) && (
     select(socket, SEL_RECV)>1))
14   // wait for incomming connection
15   {
16    buflen=select(socket, SEL_RECV);
17    // how much data have we received yet?
18    if (buflen>0)
19    {
20     nbytes=recv(socket,mydata,(buflen>mydatalen)?mydatalen:
       buflen);
21     // receive data, but not more than is avaiable and not more
        than the length of mydata
22     if (nbytes>0) send(socket,mydata,nbytes);
```

```
23      // iff we have received any data, send it back
24     }
25    }
26   }
27  }
```

Der obige Echo-Server kann nur eine einzige Anfrage beantworten. Um mehrere aufeinander folgende Anfragen beantworten zu können muss der Socket nach Abarbeitung der ersten Verbindung neu initialisiert werden. Hierzu muss er geschlossen und neu geöffnet werden. Dies geschieht mit der `close`–Funktion und der `socket`–Funktion.

Listing 8: Atmel TCP Connection Reset

```
if (data_received && !(select(socknum,SEL_CONTROL) ==
SOCK_ESTABLISHED)) {
 puts("connection closing");
 close(socket);
 wait_10ms(10);
 socket(socket,SOCK_STREAM,port,0);
 wait_10ms(10);
 NBlisten(socket);
 data_received=0;
}
```

4.3.2 TCP Client

Für einen TCP Client werden im wesentlichen die selben Funktionen benötigt, die schon für den TCP Server verwendet wurden. Zusätzlich benötigt man die `connect`–Funktion, um die Verbindung zu initiieren.

```
char connect(SOCKET s, u_char* destip, u_int port);
```

Die Funktion erhält als Parameter den Socket über den die Verbindung aufgebaut werden soll und die Ziel–IP und den Ziel–Port der Verbindung. Als Rückgabewert liefert sie 1, falls die Verbindung erfolgreich hergestellt werden konnte und -1 falls der Verbindungsaufbau fehlgeschlagen ist.

Als kleines Beispiel dient hier ein HTTP-Client: [WC]

Listing 9: Atmel HTTP Client

```
#define mydatalen 128

u_char destip[]={66,102,9,104};
// ip of e.g. google.de
u_int destport = 80;
// default http port
char* query="GET /index.html HTTP/1.1\r\nhost: google.de\r\n\r\
n";
// valid HTTP 1.1 query

int main(int argc, char* argv[]){

 /** insert socket initialization here **/

 if ((mydata=malloc(mydatalen))==NULL)
  puts("malloc -1\n"),exit(0);
 // allocate memory for the buffer

 retcode = connect(socknum,destip,destport);
 printf("done, retcode:%i\n",retcode);

 if (retcode == 1){
  send(socknum,query,strlen(data));
  while (select(socket,SEL_CONTROL) == SOCK_ESTABLISHED) {
   buflen=select(socket, SEL_RECV);
   buflen=((buflen>mydatalen)?mydatalen:buflen);
   mydata[recv(socknum,mydata,buflen)]='\0';
   printf("%s",mydata);
  }
 }
}
```

4.4 UDP

Im Gegensatz zu TCP stellt UDP weder sicher, dass ein Paket überhaupt ankommt, noch dass die Pakete in richtiger Reihenfolge empfangen werden. Der Overhead ist durch die fehlenden Acknowledgments und den kleineren Header zwar geringer, die Verwendung ist jedoch einschränkt, da UDP nur verwendet werden kann wenn der Empfänger nicht darauf angewiesen ist vollständige und fehlerfreie Daten zu empfangen. UDP ist vor allem für Video- und Audiostreaming interessant.

Ein UDP-Socket wird folgendermaßen initialisiert:

Listing 10: Atmel UDP Socket Init

```
setIPprotocol(socknum,IPPROTO_UDP);
retcode = socket(socknum,SOCK_DGRAM,port,0);
```

4.4.1 UDP Listener

Der Socket muss wie auch bei TCP auf den Status *listen* gesetzt werden. Zum Empfang der Daten wird allerdings die `recvfrom`–Funktion verwendet und nicht wie bei einem TCP Socket die `recv`–Funktion.

```
u_int recvfrom(SOCKET s, const u_char* buf, u_int len,u_char* addr, u_int
* port)
```

Die Funktion erwartet als Parameter neben dem Socket einen Pointer auf den Speicherbereich an den die Daten kopiert werden sollen, die Länge der Daten die maximal empfangen werden sollen und zwei Pointer auf Speicherbereiche, an denen die Funktion speichert von welcher Adresse und welchem Port die Daten empfangen wurden. Im Gegensatz zu einem TCP–Socket bleibt es dem Anwendungsprogrammierer überlassen, die Datenströme nach Sendern zu sortieren. Ein Vorteil ist, dass Daten von mehreren Sendern über einen Socket empfangen werden können.

Hier ein kleines Beispiel, welches die empfangenen Daten ausgibt:

Listing 11: Atmel UDP Listener

```
#define mydatalen 128

while{1} {
 if(select(socknum, SEL_RECV)) {
  nbytes=select(socknum, SEL_RECV);
  length=((nbytes>mydatalen)?mydatalen:nbytes);
  mydata[recvfrom(socknum,mydata,length,ip,&port)]='\0';
  printf("%s",mydata);
 };
}
```

Aufgrund der geringeren Verbreitung von UDP und der Tatsache, dass UDP verbindungslos arbeitet existieren weniger Programme um UDP Applikationen zu testen. Daher wird zum Testen eines UDP Listeners ein entsprechendes

Programm welches UDP Pakete sendet benötigt.

Als Beispiel ein paar Code–Zeilen, um UDP–Pakete von einem PC an eine Gegenstelle zu schicken:

Listing 12: PC UDP Sender

```
if((length=sendto(socknum,buf,sizeof(buf),0,(struct sockaddr*)&adresse,
sizeof(adresse)))<0)
 printf("Error\n");
```

Das vollständige Programm befindet sich in der Anlage dieser Studienarbeit.

4.4.2 UDP Sender

Um Daten per UDP zu senden benötigt man die `sendto`–Funktion:

```
u_int sendto(SOCKET s, const u_char* buf, u_int len, u_char* addr, u_int
port)
```

Die Parameter sind analog zur `recvfrom`–Funktion belegt. Der dritte und vierte Parameter wird jedoch hier nur gelesen und nicht verändert.

Ein kleines Beispielprogramm zur Abschätzung der maximalen UDP Sendegeschwindigkeit:

Listing 13: Atmel UDP Sender

```
while(1){
 for (i = 0; i < 10000; i++){
  sendto(socknum,mydata,strlen(mydata),destip,port);
 }
 printf("10k packages send \n");
};
```

Für aussagekräftige Testresultate wird auch hier ein PC der die Rolle der Gegenstelle übernimmt benötigt. Der entsprechende UDP–Empfänger könnte so aussehen:

Listing 14: PC UDP Listener

```
while(1){
 if((length=read(s,buf,sizeof(buf)))<0)
  printf("Error\n");
 else
  buf[length]='\0';
 printf("%s\n",buf);
}
```

5 Bibliothek

5.1 Definitionen

Die folgenden Funktionen machen intern intensiven Gebrauch von Präprozessordefinitionen[9] um den Zugriff auf den Speicher des w3100a übersichtlich zu gestalten. Daher werden zunächst die verwendeten Adressen und Konstanten erklärt.

Listing 15: `socket.h`

```
#define MAX_SOCK_NUM    4
 // maxmium number of sockets
#define I2CHIP_BASE     0x8000
 // address of w3100a
#define SEND_DATA_BUF   (volatile u_char*) 0xC000
 // internal Tx buffer address of w3100a
#define RECV_DATA_BUF   (volatile u_char*) 0xE000
 // internal Rx buffer address of w3100a

#define MAX_SEGMENT_SIZE        1460
 // Maximum TCP packet size

#define COMMAND(i)      (*((volatile u_char*)(I2CHIP_BASE + i)))
 // command register address
#define INT_STATUS(i)   (*((volatile u_char*)(I2CHIP_BASE + 0x04 + i)))
 // status register address
#define INT_REG         (*((volatile u_char*)(I2CHIP_BASE + 0x08)))
 // interrupt status register
#define INTMASK         (*((volatile u_char*)(I2CHIP_BASE + 0x09)))
 // interrupt masking register

#define RX_PTR_BASE     0x10  // address of the RX pointer
#define RX_PTR_SIZE     0x0C  // a RX pointer has the size of 4 bytes

#define RX_WR_PTR(i)    ((volatile u_char*)(I2CHIP_BASE + RX_PTR_BASE +
RX_PTR_SIZE*i))
 // calculate the RX write pointer for channel i
#define RX_RD_PTR(i)    ((volatile u_char*)(I2CHIP_BASE + RX_PTR_BASE +
RX_PTR_SIZE*i + 0x04))
 // calculate the RX read pointer for channel i
#define RX_ACK_PTR(i)   ((volatile u_char*)(I2CHIP_BASE + TX_PTR_BASE +
TX_PTR_SIZE*i + 0x08))
 // calculate the RX ack pointer for channel i

#define TX_PTR_BASE     0x40 // address of the TX pointer
#define TX_PTR_SIZE     0x0C // a TX pointer has the size of 4 bytes

#define TX_WR_PTR(i)    ((volatile u_char*)(I2CHIP_BASE + TX_PTR_BASE +
TX_PTR_SIZE*i))
 // calculate the TX write pointer for channel i
#define TX_RD_PTR(i)    ((volatile u_char*)(I2CHIP_BASE + TX_PTR_BASE +
TX_PTR_SIZE*i + 0x04))
 // calculate the TX read pointer for channel i
#define TX_ACK_PTR(i)   ((volatile u_char*)(I2CHIP_BASE + RX_PTR_BASE +
RX_PTR_SIZE*i + 0x08))
 // calculate the TX ack pointer for channel i
```

[9] auch Defines genannt

Um die validen Werte der 4 Byte langen TX/RX Schreib– und Lese–Pointer zu lesen, müssen zunächst die zugehörigen `Shadow Pointer` gelesen werden und vier Takte gewartet werden.

Listing 16: `socket.h` Fortsetzung

```
 /* Shadow Register Pointer */
#define SHADOW_RXWR_PTR(i)  ((volatile u_char*)(I2CHIP_BASE + 0x1E0 + 3*i
))
#define SHADOW_RXRD_PTR(i)  ((volatile u_char*)(I2CHIP_BASE + 0x1E1 + 3*i
))
#define SHADOW_TXACK_PTR(i) ((volatile u_char*)(I2CHIP_BASE + 0x1E2 + 3*i
))
#define SHADOW_TXWR_PTR(i)  ((volatile u_char*)(I2CHIP_BASE + 0x1F0 + 3*i
))
#define SHADOW_TXRD_PTR(i)  ((volatile u_char*)(I2CHIP_BASE + 0x1F1 + 3*i
))

#define SOCK_BASE       0xA0
#define SOCK_SIZE       0x18

#define SOCK_STATUS(i)  (*((volatile u_char*)(I2CHIP_BASE + SOCK_BASE +
SOCK_SIZE*i)))
 // status register of socket i
#define OPT_PROTOCOL(i)  (*((volatile u_char*)(I2CHIP_BASE + SOCK_BASE +
SOCK_SIZE*i + 0x01)))
 // options register of socket i
#define DST_IP_PTR(i)  ((volatile u_char*)(I2CHIP_BASE + SOCK_BASE +
SOCK_SIZE*i + 0x08))
 // pointer the destination ip address of socket i
#define DST_PORT_PTR(i)  ((volatile u_char*)(I2CHIP_BASE + SOCK_BASE +
SOCK_SIZE*i + 0x0C))
 // pointer the destination port of socket i
#define SRC_PORT_PTR(i)  ((volatile u_char*)(I2CHIP_BASE + SOCK_BASE +
SOCK_SIZE*i + 0x0E))
 // pointer the source port of socket i
#define IP_PROTOCOL(i)  (*((volatile u_char*)(I2CHIP_BASE + SOCK_BASE +
SOCK_SIZE*i + 0x10)))
 // protocol type register of socket i
#define TOS(i)  (*((volatile u_char*)(I2CHIP_BASE + SOCK_BASE + SOCK_SIZE
*i + 0x11)))
 //  the "deprecated" type of serice field of socket i
#define MSS(i)  (*((volatile u_int*)(I2CHIP_BASE + SOCK_BASE + SOCK_SIZE*
i + 0x12)))
 // maximum segment size for socket i

#define GATEWAY_PTR  ((volatile u_char*)(I2CHIP_BASE + 0x80))
 // pointer to gateway address
#define SUBNET_MASK_PTR  ((volatile u_char*)(I2CHIP_BASE + 0x84))
 // pointer to subnet mask
#define SRC_HA_PTR  ((volatile u_char*)(I2CHIP_BASE + 0x88))
 // pointer to source mac address
#define SRC_IP_PTR  ((volatile u_char*)(I2CHIP_BASE + 0x8E))
 // pointer to source ip address
#define TIMEOUT_PTR  ((volatile u_char*)(I2CHIP_BASE + 0x92))
 // pointer to timeout value

#define RX_DMEM_SIZE  (*((volatile u_char*)(I2CHIP_BASE + 0x95)))
 // bitmask of RX memory partition, see chapter 1 page 10
#define TX_DMEM_SIZE  (*((volatile u_char*)(I2CHIP_BASE + 0x96)))
 // bitmask of TX memory partition, see chapter 1 page 10
```

5.2 Speicherverwaltung des w3100a

Anhand des Beispiels aus Kapitel 4 *Chip-Initialisierung* Seite 9, in dem jeder Kanal 2KB Speicher zugewiesen bekommt, wird die Speicherverwaltung des w3100a nun näher erklärt.

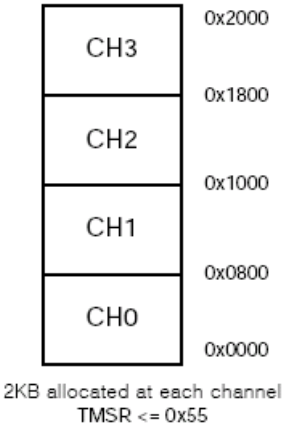


Abbildung 8: 2KB je Kanal

Der Speicher des jeweiligen Kanals (wir gehen ab jetzt der Einfachheit halber immer von Kanal 0 aus) wird als Ringpuffer angesprochen. Diese Ringpuffer werden über ihre jeweilige Basisadresse und die Puffergröße des Kanals definiert. Auf einen Sendepuffer wird über einen Write–Pointer (`TX_WR_PTR`) und einen Acknowledgment–Pointer (`TX_ACK_PTR`) zugegriffen, die jeweils den Wert der Sequenznummer des Paketes haben, auf das sie zeigen. Dieser Wert wird dann, modulo der Puffergröße, auf die Basisadresse des Puffers addiert und bildet so die reale Speicheradresse. Bei allen Operationen auf den 4Byte Pointern muss beachtet werden, dass sie in umgekehrter Reihenfolge im Speicher des w3100a stehen. Siehe hierzu das Beispiel auf Seite 9,

Listing 17: Auszüge aus `sysinit()`

```
RBUFBASEADDRESS[0] = (u_char*) RECV_DATA_BUF;

 ...

RSIZE[i] = 2048;
RMASK[i] = 0x000007FF;
```

Listing 18: Auszüge aus `recv()`

```
rd_ptr.cVal[3] = *(RX_RD_PTR(s) + 0); // read/write of 4byte
rd_ptr.cVal[2] = *(RX_RD_PTR(s) + 1); // pointer registers is slow,
rd_ptr.cVal[1] = *(RX_RD_PTR(s) + 2); // therefore we make a copy
rd_ptr.cVal[0] = *(RX_RD_PTR(s) + 3); // of them

 ...

recv_ptr = (u_char*) (RBUFBASEADDRESS[s] + (u_int)(rd_ptr.lVal & RMASK[s
]));
 // Calculate pointer to be copied received data
```

```
read_data(s, recv_ptr,(u_char*) buf, len);
```

Der Empfang von TCP Daten lässt sich also in drei Schritte gliedern.

1. Initialisierung Die Kanäle und die Receive-Read und Receive-Write-Pointer werden initialisiert (in diesem Beispiel mit dem Wert 0x00123280).

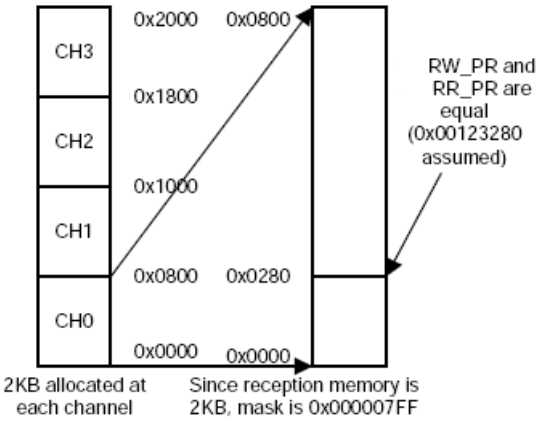


Abbildung 9: Speicherinitialisierung

2. w3100a empfängt Daten Nachdem der Socket geöffnet wurde, ist der w3100a bereit Daten zu empfangen und diese im Speicher abzulegen. Wenn er Daten empfangen hat inkrementiert er den Receive-Write-Pointer um den entsprechenden Wert.

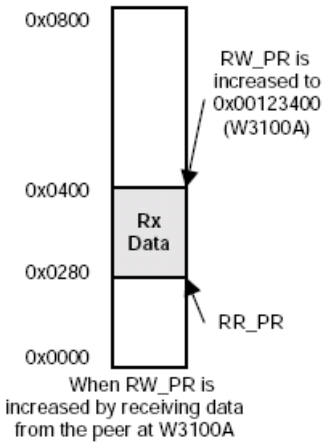


Abbildung 10: Speicherverwaltung beim Datenempfang

3. die Applikation liest die Daten Wenn nun die Applikation die Daten vom w3100a liest, wird der Receive-Read-Pointer inkrementiert bis alle Daten gelesen wurden, und die Pointer wieder gleich sind.

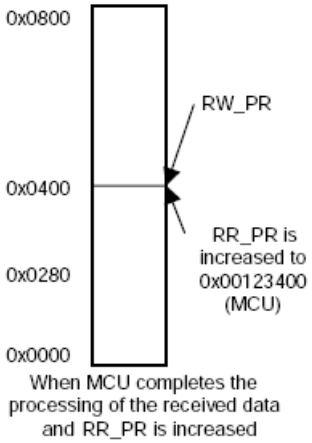


Abbildung 11: Speicherverwaltung beim Datenempfang II

5.3 API-Funktionen

5.3.1 `void initW3100A(void)`

Die Funktion `initW3100A` wird ohne Parameter aufgerufen und hat keinen Rückgabewert. Sie setzt den Standardport und die TCP Sequenznummer und führt einen Softwarereset des w3100a durch. Die TCP–Sequenznummer sollte eine Zufallszahl sein, um sicherzustellen, dass Pakete eindeutig identifiziert werden. Da Mikrocontroller in der Regel jedoch keine Möglichkeit besitzen Zufallszahlen zu erzeugen wird hier ein statischer Wert zugewiesen. Eine Auflistung der Befehlsregister des w3100a kann Kapitel 7 entnommen werden.

Listing 19: `initW3100A`

```
void initW3100A(void) {
 Local_Port = 1000;              // set default port
 SEQ_NUM.lVal = 0x9115F3F1;      // sets the initial SEQ#
 COMMAND(0) = CSW_RESET;         // Software RESET
}
```

5.3.2 `void setIP(u_char* addr)`

Die Funktion `setIP` schreibt die IP-Adresse, die ihr mittels eines Pointers auf ein *u_char*–Feld übergeben wird byteweise an die Adresse auf die `SRC_IP_PTR` zeigt. Dort wird sie vom w3100a gelesen und als Source–IP verwendet.

Listing 20: `setIP`

```
void setIP(u_char* addr){
 for (u_char i = 0; i < 4; i++){
```

```
  *(SRC_IP_PTR + i) = addr[i];
 }
}
```

5.3.3 void setsubmask(u_char* addr)

Die Funktion `setsubmask` schreibt die Subnet–Maske, die mittels eines Pointers auf ein *u_char*–Feld übergeben wird byteweise an die Adresse auf die `SUBNET_MASK_PTR` zeigt. Dort wird sie vom w3100a gelesen und als Subnet–Maske verwendet.

Listing 21: `setsubmask`

```
void setsubmask(u_char* addr){
 for (u_char i = 0; i < 4; i++){
  *(SUBNET_MASK_PTR + i) = addr[i];
 }
}
```

5.3.4 void setgateway(u_char* addr)

Die Funktion `setgateway` schreibt die Adresse des Standardgateways, die mittels eines Pointers auf ein *u_char*–Feld übergeben wird byteweise an die Adresse auf die `GATEWAY_PTR` zeigt. Dort wird sie vom w3100a gelesen und als Gateway für alle IP–Pakete verwendet die nicht an Ziele innerhalb des lokalen Netzes, welches durch IP–Adresse und Subnet-Maske definiert wird, gesendet werden.

Listing 22: `setgateway`

```
void setgateway(u_char* addr){
 for (u_char i = 0; i < 4; i++){
  *(GATEWAY_PTR + i) = addr[i];
 }
}
```

5.3.5 void setMACAddr(u_char* addr)

Die Funktion `setMACAddr` schreibt die MAC-Adresse, die mittels eines Pointers auf ein *u_char*–Feld übergeben wird byteweise an die Adresse auf die `SRC_HA_PTR` zeigt. Die MAC–Adresse ist hardwaregebunden und sollte bei jedem ethernetfähigen Gerät weltweit einzigartig sein. Sie steht normalerweise im EPROM der Netzwerkkarte, muss aber beim w3100a per Software gesetzt werden.

Listing 23: `setMACAddr`

```
void setMACAddr(u_char* addr){
 for (u_char i = 0; i < 6; i++){
  *(SRC_HA_PTR + i) = addr[i];
 }
}
```

5.3.6 void setIPprotocol(SOCKET s, u_char IP_PROTOCOL)

Die Funktion `setIPprotocol` schreibt den Protokolltyp des Sockets *s*, an die Stelle die durch die Makro[10] `IP_PROTOCOL` definiert wird (die gültigen Werte hierfür können Listing 4 auf Seite 11 entnommen werden). Die Wahl des Protokolltyps hat keinen Einfluss auf die interne Arbeitsweise des w3100a, sondern setzt Wert des Protokoll–Felds in den ausgehenden IP–Paketen.

Listing 24: `setIPprotocol`

```
void setIPprotocol(SOCKET s, u_char ipprotocol){
 IP_PROTOCOL(s) = ipprotocol;
}
```

5.3.7 void settimeout(u_char* val)//OPTIONAL

Die Funktion `settimeout` setzt sowohl das Register IRTR (Initial Retry Timevalue Register) als auch RCR (Retry Count Register) des W3100a anhand der Werte die im übergebenen *u_char*–Feld stehen. Diese Register werden vom w3100a verwendet um zu entscheiden, nach wie vielen Mikrosekunden ein TCP–Paket erneut übertragen wird, falls das Acknowledgment ausbleibt, und nach wie vielen Versuchen die Übertragung abgebrochen wird. Der Wert im Register IRTR entspricht der Wartezeit in Mikrosekunden (`03 F8` entspricht der Wartezeit von 100ms).

Listing 25: `settimeout`

```
void settimeout(u_char* val){
 for (u_char i = 0; i < 3; i++) {
  *(TIMEOUT_PTR + i) = val[i];
 }
}
```

5.3.8 void setTOS(SOCKET s, u_char tos)//OPTIONAL

Die Funktion `setTOS` schreibt den `Type of Service` der IP Pakete die über den Socket *s* versendet werden an die Speicherstelle die durch das Makro `TOS` vorgegeben wird. Der `Type of Service` wurde ursprünglich benutzt um die IP Pakete in drei Prioritätsklassen einzuteilen, wird nun jedoch für das DSCP/ECN Protokoll verwendet, welches jedoch nicht von vielen Geräten unterstützt wird.[Ram]

Listing 26: `setTOS`

```
void setTOS(SOCKET s, u_char tos){
 TOS(s) = tos;
}
```

5.3.9 void sysinit(u_char sbufsize, u_char rbufsize)

Die Funktion `sysinit` erhält wie schon bereits erwähnt als Parameter zwei Bitfolgen, die die Verteilung des Speichers auf die Sende– und Empfangspuffer

[10] definiert auf Seite 19

angeben. Aus den Bitfolgen werden über *switch*–Strukturen für jeden Kanal zwei Bitmasken generiert mit denen später einfach die physischen Adressen der Daten im virtuellen Ringpuffer berechnet werden und die Größe der jeweiligen Ringpuffer berechnet. Aus der Größe der Ringpuffer werden dann die Basisadressen der physischen Speicherbereiche berechnet. Da noch keine Daten empfangen oder versendet wurden, werden die Pointer für die Sende– und Empfangspuffer mit 0 initialisiert. Dem w3100a-Chip wird dann mitgeteilt, dass alle für die Initialisierung notwendigen Werte gesetzt wurden und es wird darauf gewartet, dass der Chip die erfolgreiche Initialisierung meldet.

Listing 27: `sysinit`

```
1  void sysinit(u_char sbufsize, u_char rbufsize) {
2   u_char i=0;
3
4   int ssum = 0;
5   int rsum = 0;
6
7   TX_DMEM_SIZE = sbufsize;                // Set Tx memory size
    for each channel
8   RX_DMEM_SIZE = rbufsize;                // Set Rx memory size
    for each channel
9
10  SBUFBASEADDRESS[0] = (u_char*) SEND_DATA_BUF;  // Set Base
    Address of Tx memory for channel #0
11  RBUFBASEADDRESS[0] = (u_char*) RECV_DATA_BUF;  // Set Base
    Address of Rx memory for channel #0
12
13  for(i = 0 ; i < MAX_SOCK_NUM; i++) {   // for all channels do:
14   SSIZE[i] = 0;
15   RSIZE[i] = 0;
16
17   if(ssum < 8192) {
18    switch((sbufsize>> i*2) & 0x03) {    // Set maximum Tx
      memory size
19     case 0:
20      SSIZE[i] = 1024;
21      SMASK[i] = 0x000003FF;
22      break;
23     case 1:
24      SSIZE[i] = 2048;
25      SMASK[i] = 0x000007FF;
26      break;
27     case 2:
28      SSIZE[i] = 4096;
29      SMASK[i] = 0x00000FFF;
30      break;
31     case 3:
32      SSIZE[i] = 8192;
33      SMASK[i] = 0x00001FFF;
34      break;
35    }
36   }
37
38   if(rsum < 8192) {
39    switch((rbufsize>> i*2) & 0x03) { // Set maximum Rx memory
      size
40     case 0:
41      RSIZE[i] = 1024;
```

```
42        RMASK[i] = 0x000003FF;
43        break;
44       case 1:
45        RSIZE[i] = 2048;
46        RMASK[i] = 0x000007FF;
47        break;
48       case 2:
49        RSIZE[i] = 4096;
50        RMASK[i] = 0x00000FFF;
51        break;
52       case 3:
53        RSIZE[i] = 8192;
54        RMASK[i] = 0x00001FFF;
55        break;
56     }
57    }
58
59    ssum += SSIZE[i];
60    rsum += RSIZE[i];
61
62    if(i != 0) {  // Set base address of Tx and Rx memory for
      channel #1,#2,#3
63     SBUFBASEADDRESS[i] = SBUFBASEADDRESS[i-1] + SSIZE[i-1];
64     RBUFBASEADDRESS[i] = RBUFBASEADDRESS[i-1] + RSIZE[i-1];
65    }
66    *((u_long*) TX_WR_PTR(i))=0;  // Transmit Buffer Write
      Pointer initialization of channel #i
67    *((u_long*) TX_RD_PTR(i))=0;  // Transmit Buffer Read Pointer
       initialization of channel #i
68    *((u_long*) RX_RD_PTR(i))=0;  // Receive Buffer Read Pointer
      initialization of channel #i
69    *((u_long*) RX_WR_PTR(i))=0;  // Receive Buffer Write Pointer
       initialization of channel #i
70   }
71
72   COMMAND(0) = CSYS_INIT;
73   I_STATUS[0] = INT_STATUS(0);
74
75   while(!(I_STATUS[0]& SSYS_INIT_OK)){ // wait until w3100a is
     ready
76    I_STATUS[0] = INT_STATUS(0);
77   }
78  }
```

5.3.10 char socket(SOCKET s, u_char protocol, u_int port, u_char flag)

Die Funktion `socket` erhält als Parameter die *SOCKET*-Nummer, den Protokolltyp, den Source-Port und die Optionen für den *SOCKET*. Falls ein Port ungleich 0 übergeben wurde, wird dieser als Source-Port verwendet. Falls nicht wird der in der `initW3100A`–Funktion gesetzte Standard-Port `Local_Port` inkrementiert und als Source-Port verwendet. Hierbei muss die Byte-Order mit der `swapuint`–Funktion konvertiert werden, da der w3100a den Port in Big-Endian Byte-Order erwartet.
Danach werden die Status-Register zurückgesetzt und die Initialisierung des w3100a gestartet. Die Initialisierung des w3100a ist abgeschlossen, wenn das

INT_STATUS–Register den Wert 0 hat. Falls die Initialisierung nicht erfolgreich war wird −1 zurückgegeben, ansonsten wird die Sequenznummer initialisiert und die Socketnummer zurückgegeben.

Listing 28: socket

```
1  char socket(SOCKET s, u_char protocol, u_int port, u_char flag)
   {
2   OPT_PROTOCOL(s) = protocol | flag;
3    // set socket protocol and options
4
5   if (port != 0) {       // user specified or lib-choosen port
6    *(SRC_PORT_PTR(s)) = swapuint(port); // user specified
7   }
8   else {
9    Local_Port++;         // use the next port as source port
10   *SRC_PORT_PTR(s) = swapuint(Local_Port);
11  }
12
13  I_STATUS[s] = 0;       // reset status
14  INT_STATUS(s) = 0;     // reset status
15  COMMAND(s) = CSOCK_INIT;  // tell the w3100a chip to init
16
17  I_STATUS[s]=INT_STATUS(s);     // wait until the w3100a
18  while (I_STATUS[s] == 0){      // chip has finished
19   I_STATUS[s]=INT_STATUS(s);    // the initialization
20   wait_10ms(1);                 //
21  };
22
23  if (!(I_STATUS[s] & SSOCK_INIT_OK)) return (-1);
24   // initialization has failed
25
26  initseqnum(s); // set a "random" number for seq#
27  return(s);
28 }
```

5.3.11 void initseqnum(SOCKET s)

Die Funktion initseqnum setzt die initiale Sequenznummer des *Socket*s s und initialisiert die read-, write- und ack-Pointer entsprechend. Um zu verstehen wofür dies notwendig ist muss man verstehen wie der w3100A seinen Empfangs- und Sendepuffer verwaltet. Dies wird im Abschnitt 5.2 Speicherverwaltung Seite 20 näher erklärt.

Listing 29: initseqnum

```
1  void initseqnum(SOCKET s) {
2   SEQ_NUM.lVal++;        // this should be a random value
3
4   *(TX_WR_PTR(s) + 0) = SEQ_NUM.cVal[3]; // unintuitive byte
5   *(TX_WR_PTR(s) + 1) = SEQ_NUM.cVal[2]; // order because of
6   *(TX_WR_PTR(s) + 2) = SEQ_NUM.cVal[1]; // big/little endian
7   *(TX_WR_PTR(s) + 3) = SEQ_NUM.cVal[0]; // conversion
8
9   wait_1us(2);   // Wait until TX_WR_PRT has been written safely
```

```
10
11   *(TX_RD_PTR(s) + 0) = SEQ_NUM.cVal[3];
12   *(TX_RD_PTR(s) + 1) = SEQ_NUM.cVal[2];
13   *(TX_RD_PTR(s) + 2) = SEQ_NUM.cVal[1];
14   *(TX_RD_PTR(s) + 3) = SEQ_NUM.cVal[0];
15
16   wait_1us(2);   // Wait until TX_RD_PRT has been written safely
17
18   *(TX_ACK_PTR(s) + 0) = SEQ_NUM.cVal[3];
19   *(TX_ACK_PTR(s) + 1) = SEQ_NUM.cVal[2];
20   *(TX_ACK_PTR(s) + 2) = SEQ_NUM.cVal[1];
21   *(TX_ACK_PTR(s) + 3) = SEQ_NUM.cVal[0];
22  }
```

5.3.12 char connect(SOCKET s, u_char* addr, u_int port)

Die Funktion `connect` initiiert als Client eine TCP Verbindung mit einem Server. Sie erhält als Parameter eine *SOCKET*–Nummer, einen Pointer auf die Zieladresse und einen Zielport. Da eine TCP Verbindung immer nur zu einem TCP Server aufgebaut wird, der auf einem bestimmten Port auf eine Verbindung wartet, wird der Verbindungsaufbau abgebrochen falls kein valider Port übergeben wurde, sonst wird der übergebene Port an den w3100a übergeben. Nun wird der Status zurückgesetzt und der Befehl zum Verbindungsaufbau an den w3100a übergeben. (*Zeile 16*)
Nachdem der w3100a den Versuch, die Verbindung aufzubauen abgeschlossen hat, wird überprüft ob die Verbindung zu Stande kam. Falls dies nicht der Fall ist wird der *SOCKET* wieder geschlossen. Wenn alles wie beabsichtigt geklappt hat, und die Verbindung hergestellt ist, wird ein 1 an die aufrufende Funktion zurückgegeben.

Listing 30: `connect`

```
1  char connect(SOCKET s, u_char* addr, u_int port)
2  {
3   if (port != 0) {
4    *(DST_PORT_PTR(s) = swapuint(port);
5   }
6   else {          // we need a dest port,
7    return (-1);  // so lets break if none is provided
8   }
9
10  *(DST_IP_PTR(s) + 0) = addr[0]; // set dest ip
11  *(DST_IP_PTR(s) + 1) = addr[1];
12  *(DST_IP_PTR(s) + 2) = addr[2];
13  *(DST_IP_PTR(s) + 3) = addr[3];
14
15  I_STATUS[s] = 0;
16  COMMAND(s) = CCONNECT; // connect!
17
18  I_STATUS[s]=INT_STATUS(s);
19
20  while (I_STATUS[s] == 0) {
21   I_STATUS[s]=INT_STATUS(s);
22   wait_1ms(10); // wait until connection init has finished
23  }
```

```
24
25   if (select(s, SEL_CONTROL) == SOCK_CLOSED){
26    return (-1);  // when failed, close channel and return an
      error
27   }
28
29   I_STATUS[s]=INT_STATUS(s);
30
31   if (!(I_STATUS[s] & SESTABLISHED)) {
32    return (-1);  // something else has gone wrong
33   }
34
35   return (1);     // return 1, if everything is ok
36  }
```

5.3.13 void NBlisten(SOCKET s)

Die Funktion `NBlisten` arbeitet nicht-blockierend. Sie setzt den *SOCKET s* in den Modus *LISTENING*, aktualisiert die Statusvariable und kehrt zur aufrufenden Funktion zurück.

Listing 31: `NBlisten`

```
void NBlisten(SOCKET s) {
 COMMAND(s) = CLISTEN;           // set socket to "LISTENING"
 I_STATUS[s]=INT_STATUS(s);
}
```

5.3.14 int send(SOCKET s, const u_char* buf, u_int len)

Die Funktion `send` ist eine Art Wrapper-Funktion für die interne `send_in`–Funktion. Sie erhält als Parameter den Socket über den die Daten gesendet werden sollen, einen Pointer auf den Bereich in dem die zu sendenden Daten liegen und die Länge der Daten. Die Funktion überprüft ob Daten zu senden sind und ruft dann so lange die `send_in`–Funktion auf bis alle Daten gesendet wurden oder ein Fehler auftritt. Am Ende wird die Menge der insgesamt gesendeten Daten zurückgegeben.

Listing 32: `send`

```
1  int send(SOCKET s, const u_char* buf, u_int len) {
2   int ptr, size;
3
4   if (len <= 0)
5    return (0);
6   else {
7    ptr = 0;
8     while (len > 0) {
9      size = send_in(s, buf + ptr, len);
10     if (size == -1) return -1;
11     len = len - size;
12     ptr += size;
13    }
14  }
```

```
15   return ptr;
16  }
```

5.3.15 int recv(SOCKET s, const u_char* buf, u_int len)

Die Funktion recv liest die Daten, die sich zur Zeit im Empfangspuffer des TCP Sockets *s* befinden, schreibt sie in den Puffer *buf* und gibt die Menge der gelesenen Bytes zurück. Der Parameter *len* gibt an, wieviele Daten maximal gelesen werden sollen, und hat i.d.R. den Wert der Länge von *buf*. Die Funktion liest zunächst die Schreib- und Lesepointer aus dem Speicher des w3100a und berechnet aus ihnen, wieviele Daten sich im Puffer befinden. Falls bis jetzt weniger Daten als *len* empfangen wurden, wartet die Funktion bis genügend Daten empfangen wurden. Dann wird die Funktion *read_data* aufgerufen um die Daten vom Speicher des w3100a in den Speicher des Atmel zu kopieren. Anschließend wird der Lesepointer auf dem w3100a aktualisiert und die Funktion ist beendet.

Listing 33: recv

```
1  int recv(SOCKET s, const u_char* buf, u_int len) {
2   u_char k;
3   u_int size;
4   un_l2cval wr_ptr, rd_ptr;
5   u_char* recv_ptr;
6
7   R_START:
8    cli();
9    k = *SHADOW_RXWR_PTR(s); // Must read the shadow register for
      reading 4byte pointer registers
10   wait_1us(2); // wait for reading 4byte pointer registers
     safely
11
12   wr_ptr.cVal[3] = *(RX_WR_PTR(s) + 0);
13   wr_ptr.cVal[2] = *(RX_WR_PTR(s) + 1);
14   wr_ptr.cVal[1] = *(RX_WR_PTR(s) + 2);
15   wr_ptr.cVal[0] = *(RX_WR_PTR(s) + 3);
16
17   k = *SHADOW_RXRD_PTR(s); // Must read the shadow register for
      reading 4byte pointer registers
18   wait_1us(2); // wait for reading 4byte pointer registers
     safely
19
20   rd_ptr.cVal[3] = *(RX_RD_PTR(s) + 0);
21   rd_ptr.cVal[2] = *(RX_RD_PTR(s) + 1);
22   rd_ptr.cVal[1] = *(RX_RD_PTR(s) + 2);
23   rd_ptr.cVal[0] = *(RX_RD_PTR(s) + 3);
24   sei();
25
26    // calculate received data size
27   if ( len <= 0 )
28    return (0);
29   else
30    if (wr_ptr.lVal >= rd_ptr.lVal)
31     size = (u_int) (wr_ptr.lVal - rd_ptr.lVal);
32    else
33     size= (u_int)(0 - (rd_ptr.lVal - wr_ptr.lVal));
```

```
34
35   if (size < len) {   // Wait until receiving is done when
     received data size is less then len
36    if (select(s, SEL_CONTROL) != SOCK_ESTABLISHED) return -1;
            // Error
37
38   wait_1ms(3);
39   goto R_START;
40  }
41  recv_ptr = (u_char*) (RBUFBASEADDRESS[s] + (UINT)(rd_ptr.lVal
    & RMASK[s]));
42   // Calculate pointer to be copied received data
43
44  read_data(s, recv_ptr,(u_char*) buf, len); // Copy received
    data
45
46  rd_ptr.lVal += len;
47
48  *(RX_RD_PTR(s) + 0) = rd_ptr.cVal[3];
49  *(RX_RD_PTR(s) + 1) = rd_ptr.cVal[2];
50  *(RX_RD_PTR(s) + 2) = rd_ptr.cVal[1];
51  *(RX_RD_PTR(s) + 3) = rd_ptr.cVal[0];
52
53  COMMAND(s) = CRECV; // RECV
54  return (len);
55 }
```

5.3.16 u_int sendto(SOCKET s, const u_char* buf, u_int len, u_char* addr, u_int port)

Die Funktion `sendto` ist eine Art Wrapper Funktion für die interne `sendto_in`–Funktion und arbeitet analog zur *send*–Funktion. Der Hauptunterschied ist, dass durch die Verwendung des verbindungslosen UDP Protokolls bei jedem Aufruf Zieladresse und Zielport übergeben werden müssen.

Listing 34: `sendto`

```
1  u_int sendto(SOCKET s, const u_char* buf, u_int len, u_char*
   addr, u_int port) {
2   int ptr, size;
3
4
5   while(COMMAND(s) & CSEND) { // Wait until previous send
    command has completed
6    if(select(s,SEL_CONTROL) == SOCK_CLOSED) return -1;
7   };
8
9   if (port != 0) { // set port number
10   port=swapuint(port); // endian conversion
11   *(u_int*)(DST_PORT_PTR(s)) = port;
12  }
13
14  *(DST_IP_PTR(s) + 0) = addr[0];
15  *(DST_IP_PTR(s) + 1) = addr[1];
16  *(DST_IP_PTR(s) + 2) = addr[2];
17  *(DST_IP_PTR(s) + 3) = addr[3];
18   // set destination IP address
```

```
19
20   if (len <= 0)
21    return (0);
22   else {
23    ptr = 0;
24    while ( len > 0) {
25     size = sendto_in(s, buf + ptr, len);
26     if(size == -1) return -1; // Error
27     len = len - size;
28     ptr += size;
29    };
30   }
31   return ptr;
32  }
```

5.3.17 u_int recvfrom(SOCKET s, const u_char* buf, u_int len, u_char* addr, u_int* port)

Die Funktion `recvfrom` liest die Daten, die sich zur Zeit im Empfangspuffer des UDP Sockets *s* befinden, schreibt sie in den Puffer *buf* und gibt die Menge der gelesenen Bytes zurück. Der Parameter *len* gibt an, wieviele Daten maximal gelesen werden sollen, und hat i.d.R. den Wert der Länge von *buf*. Da UDP verbindungslos arbeitet muss ein Teil der Aufgaben die bei Verwendung von TCP der w3100a übernimmt, von der Bibliothek bzw. vom Hauptprogramm übernommen werden.
Deutlich wird dies vor allem an der Verwendung des `_UDPHeader`–Structs, welches dazu verwendet wird, den UDP–Header, der innerhalb des Empfangspuffer liegt zu parsen. Die im Header stehende Adresse und Port des Senders wird im *port*– und *addr*–Array abgelegt und kann vom aufrufenden Programm ausgewertet werden. Ansonsten arbeitet die Funktion analog zur `recv`–Funktion.

Listing 35: `recvfrom`

```
1  u_int recvfrom(SOCKET s, const u_char* buf, u_int len,u_char*
   addr, u_int* port) {
2   struct _UDPHeader {
3    // When receiving UDP data, we need to parse the header
     manually
4    union {
5     struct {
6      u_int size;
7      u_char addr[4];
8      u_int port;
9     } header;
10    u_char stream[8];
11   } u;
12  } UDPHeader;
13
14  u_int ret=0;
15  u_char* recv_ptr;
16  un_l2cval wr_ptr, rd_ptr;
17  u_long size;
18  u_char k;
19
20  if(select(s,SEL_CONTROL)==SOCK_CLOSED) return -1;
```

```
21
22  cli();
23  k = *SHADOW_RXWR_PTR(s);
24  wait_1us(2);
25  wr_ptr.cVal[3] = *(RX_WR_PTR(s) + 0);
26  wr_ptr.cVal[2] = *(RX_WR_PTR(s) + 1);
27  wr_ptr.cVal[1] = *(RX_WR_PTR(s) + 2);
28  wr_ptr.cVal[0] = *(RX_WR_PTR(s) + 3);
29
30  k = *SHADOW_RXRD_PTR(s);
31  wait_1us(2);
32  rd_ptr.cVal[3] = *(RX_RD_PTR(s) + 0);
33  rd_ptr.cVal[2] = *(RX_RD_PTR(s) + 1);
34  rd_ptr.cVal[1] = *(RX_RD_PTR(s) + 2);
35  rd_ptr.cVal[0] = *(RX_RD_PTR(s) + 3);
36  sei();
37
38 // Calculate received data size
39
40  if(len <=0)
41   return 0;
42  else
43   if (wr_ptr.lVal >= rd_ptr.lVal)
44    size = (u_int) wr_ptr.lVal - rd_ptr.lVal;
45   else
46    size = (u_int)(0 - (rd_ptr.lVal - wr_ptr.lVal));
47
48  if (size == 0) return 0;
49
50  recv_ptr = (u_char*)(RBUFBASEADDRESS[s] + (UINT)(rd_ptr.lVal &
     RMASK[s]));
51   // Calulate received data pointer
52
53  if ((OPT_PROTOCOL(s) & 0x07) == SOCK_DGRAM) {
54   read_data(s,recv_ptr,UDPHeader.u.stream,8);
55    // w3100a UDP header copy
56
57   addr[0] = UDPHeader.u.header.addr[0];
58   addr[1] = UDPHeader.u.header.addr[1];
59   addr[2] = UDPHeader.u.header.addr[2];
60   addr[3] = UDPHeader.u.header.addr[3];
61    // Read Source IP address
62
63   *port = UDPHeader.u.stream[6];
64   *port = (*port << 8) + UDPHeader.u.stream[7];
65    // Read Source port address
66
67   size =  swapuint(UDPHeader.u.header.size)-8;
68
69   rd_ptr.lVal += 8;
70   recv_ptr = (u_char*)(RBUFBASEADDRESS[s] + (UINT)(rd_ptr.lVal
     & RMASK[s]));
71   ret = read_data(s, recv_ptr, (u_char*) buf, size);
72
73   rd_ptr.lVal += ret;
74  }
75  *(RX_RD_PTR(s) + 0) = rd_ptr.cVal[3];
76  *(RX_RD_PTR(s) + 1) = rd_ptr.cVal[2];
77  *(RX_RD_PTR(s) + 2) = rd_ptr.cVal[1];
78  *(RX_RD_PTR(s) + 3) = rd_ptr.cVal[0];
79
80  COMMAND(s) = CRECV;
```

```
81   return (ret);  // return received data size
82  }
```

5.3.18 void close(SOCKET s)

Die Funktion close schließt den Socket, der ihr übergeben wird. Falls sich der Socket schon im Zustand SOCK_CLOSED befindet, ist nichts zu tun und die Funktion wird beendet. Falls dies nicht der Fall ist, wird überprüft, ob alle zu sendenden Daten gesendet wurden. Dies ist der Fall, wenn die Größe des freien Sendepuffers gleich der Größe des gesamten Sendepuffers ist. Nun wird dem w3100a der Befehl zum Schließen des Sockets gegeben und gewartet bis dieser diesen Befehl abgeschlossen hat.

Listing 36: close

```
1  void close(SOCKET s) {
2   if (select(s, SEL_CONTROL) == SOCK_CLOSED) return; // Already
    closed
3
4   if (select(s, SEL_SEND)== SSIZE[s]) {
5    I_STATUS[s]=INT_STATUS(s);
6    COMMAND(s) = CCLOSE;
7    while(!(I_STATUS[s] & SCLOSED)){
8     I_STATUS[s]=INT_STATUS(s);
9     wait1us(10);
10   }
11  }
12 }
```

5.3.19 u_int select(SOCKET s, u_char func)

Die Funktion select ist eine Funktion, die Statusinformationen über den ihr übergebenen Socket ausgibt. Die Funktion kennt drei Befehle, sie kann den Socketstatus und sowohl die Größe des freien Sende- als auch des freien Empfangspuffers ausgeben. Zur Ermittlung des Socketstatus wird das entsprechende Register ausgelesen und der Wert zurückgegeben. Die Größe der Puffer ist jeweils gleich der Differenz der Read- und Writepointer.

Listing 37: select

```
1  u_int select(SOCKET s, u_char func) {
2   u_int val=0;
3   un_l2cval rd_ptr, wr_ptr, ack_ptr;
4   u_char k;
5
6   switch (func) {
7    case SEL_CONTROL:
8    // socket status information
9      return (SOCK_STATUS(s));
10
```

```
11    case SEL_SEND:
12    // Calculate send free buffer size
13      cli();
14      k = *SHADOW_TXWR_PTR(s);
15      wait_1us(2);
16      wr_ptr.cVal[3] = *(TX_WR_PTR(s) + 0);
17      wr_ptr.cVal[2] = *(TX_WR_PTR(s) + 1);
18      wr_ptr.cVal[1] = *(TX_WR_PTR(s) + 2);
19      wr_ptr.cVal[0] = *(TX_WR_PTR(s) + 3);
20
21      if( (OPT_PROTOCOL(s)& 0x07) != SOCK_STREAM) {
22       k = *SHADOW_TXRD_PTR(s);
23       wait_1us(2);
24       ack_ptr.cVal[3] = *(TX_RD_PTR(s) + 0);
25       ack_ptr.cVal[2] = *(TX_RD_PTR(s) + 1);
26       ack_ptr.cVal[1] = *(TX_RD_PTR(s) + 2);
27       ack_ptr.cVal[0] = *(TX_RD_PTR(s) + 3);
28      }
29      else {
30       k = *SHADOW_TXACK_PTR(s);
31       wait_1us(2);
32       ack_ptr.cVal[3] = *(TX_ACK_PTR(s) + 0);
33       ack_ptr.cVal[2] = *(TX_ACK_PTR(s) + 1);
34       ack_ptr.cVal[1] = *(TX_ACK_PTR(s) + 2);
35       ack_ptr.cVal[0] = *(TX_ACK_PTR(s) + 3);
36      }
37      sei();
38
39      if (wr_ptr.lVal >= ack_ptr.lVal)
40       val = SSIZE[s] -  (u_int) (wr_ptr.lVal - ack_ptr.lVal);
41      else
42       val=  SSIZE[s] - (u_int)(0 - (rd_ptr.lVal - wr_ptr.lVal));
43
44      return (val);
45
46    case SEL_RECV:
47    // Calculate received data size
48      cli();
49      k = *SHADOW_RXWR_PTR(s);
50      wait_1us(2);
51      wr_ptr.cVal[3] = *(RX_WR_PTR(s));
52      wr_ptr.cVal[2] = *(RX_WR_PTR(s) + 1);
53      wr_ptr.cVal[1] = *(RX_WR_PTR(s) + 2);
54      wr_ptr.cVal[0] = *(RX_WR_PTR(s) + 3);
55
56      k = *SHADOW_RXRD_PTR(s);
57      wait_1us(2);
58      rd_ptr.cVal[3] = *(RX_RD_PTR(s) + 0);
59      rd_ptr.cVal[2] = *(RX_RD_PTR(s) + 1);
60      rd_ptr.cVal[1] = *(RX_RD_PTR(s) + 2);
61      rd_ptr.cVal[0] = *(RX_RD_PTR(s) + 3);
62
63      sei();
64
65      if (wr_ptr.lVal == rd_ptr.lVal){
66       val = 0;
67      }
68      else
69       if (wr_ptr.lVal > rd_ptr.lVal)
70        val = wr_ptr.lVal - rd_ptr.lVal;
71       else
72        val = (u_int)(0 - (rd_ptr.lVal - wr_ptr.lVal));
```

```
73
74      return (val);
75
76    default:
77    // unknown command
78      return (-1);
79   }
80   return (val);
81 }
```

5.4 Interne Funktionen

5.4.1 int send_in(SOCKET s, const u_char* buf, u_int len)

Die Funktion `send_in` sorgt dafür, dass (maximal *len*) Daten aus dem Puffer *buf* in den Sendepuffer des w3100a kopiert werden. Hierzu werden die Write- und Acknowledgment-Pointer gelesen, da Daten, für die noch kein Acknowledgment eingetroffen ist, nicht überschrieben werden dürfen. Aus diesen Pointern wird die Größe des freien Speicherbereichs berechnet. Nach der Überprüfung des Socketstatus werden dann, soviele Daten wie möglich in den Sendepuffer kopiert, indem die Funktion `write_data` aufgerufen wird. Am Ende wird der Write-Pointer aktualisiert und die Menge der wirklich geschriebenen Daten zurückgegeben.

Listing 38: `send_in`

```
1  int send_in(SOCKET s, const u_char* buf, u_int len) {
2   u_char k;
3   int size;
4   un_l2cval wr_ptr, ack_ptr;
5   u_char* send_ptr;
6
7   S_START:
8    cli();
9    k = *SHADOW_TXWR_PTR(s);
10    // Must read the shadow register before reading 4byte
      pointer registers
11   wait_1us(2);
12    // wait for reading 4byte pointer registers safely
13   wr_ptr.cVal[3] = *(TX_WR_PTR(s) + 0);
14   wr_ptr.cVal[2] = *(TX_WR_PTR(s) + 1);
15   wr_ptr.cVal[1] = *(TX_WR_PTR(s) + 2);
16   wr_ptr.cVal[0] = *(TX_WR_PTR(s) + 3);
17
18   k = *SHADOW_TXACK_PTR(s);
19   wait_1us(2);
20   ack_ptr.cVal[3] = *(TX_ACK_PTR(s) + 0);
21   ack_ptr.cVal[2] = *(TX_ACK_PTR(s) + 1);
22   ack_ptr.cVal[1] = *(TX_ACK_PTR(s) + 2);
23   ack_ptr.cVal[0] = *(TX_ACK_PTR(s) + 3);
24   sei();
25
26   // Calculate send free buffer size
27   if (wr_ptr.lVal >= ack_ptr.lVal)
28    size = SSIZE[s] - (u_int) (wr_ptr.lVal - ack_ptr.lVal);
29   else
```

```
30      size=  SSIZE[s] - (u_int) (0 - (ack_ptr.lVal - wr_ptr.lVal))
        ;
31    if (size > SSIZE[s]) {
32      // Recalulate after some delay
33     if (select(s, SEL_CONTROL) != SOCK_ESTABLISHED) return -1;
       // Error
34     wait_1ms(1);
35     goto S_START;
36    }
37
38   if (size == 0) { // Wait when previous sending has not
     finished yet and there's no free buffer
39    if (select(s, SEL_CONTROL) != SOCK_ESTABLISHED) return -1; //
       Error
40    wait_1ms(1);
41    goto S_START;
42   }
43   else {
44    if (size < len) len = size;}
45
46   send_ptr = (u_char*)( SBUFBASEADDRESS[s] + (UINT)(wr_ptr.lVal
     & SMASK[s]));
47    // Calculate pointer to copy data
48
49   write_data(s, (u_char*) buf, send_ptr, len);
50    // copy data
51
52   while (COMMAND(s) & CSEND) // Confirm send command
53   if (select(s, SEL_CONTROL) != SOCK_ESTABLISHED) return -1; //
     Error
54
55   wr_ptr.lVal = wr_ptr.lVal + len;
56    // tx_wr_ptr update
57
58   *(TX_WR_PTR(s) + 0) = wr_ptr.cVal[3];
59   *(TX_WR_PTR(s) + 1) = wr_ptr.cVal[2];
60   *(TX_WR_PTR(s) + 2) = wr_ptr.cVal[1];
61   *(TX_WR_PTR(s) + 3) = wr_ptr.cVal[0];
62
63   COMMAND(s) = CSEND; // SEND
64   return(len);
65  }
```

5.4.2 u_int sendto_in(SOCKET s, const u_char* buf, u_int len)

Die Funktion sendto_in arbeitet analog zur send_in–Funktion, jedoch nur auf UDP Sockets. Da es im UDP Protokoll keine Acknowledgments gibt, muss hierbei jedoch nur der Read-Pointer und nicht der Acknowledgment-Pointer beachtet werden.

Listing 39: sendto_in

```
1  u_int sendto_in(SOCKET s, const u_char* buf, u_int len) {
2   u_char k;
3   u_int size;
4   un_l2cval wr_ptr, rd_ptr;
5   u_char* send_ptr;
```

```
6
7   S2_START:
8    if(select(s,SEL_CONTROL)==SOCK_CLOSED) return -1;     //
     Error
9
10   cli();
11   k = *SHADOW_TXWR_PTR(s);
12   wait_1us(2);
13   wr_ptr.cVal[3] = *(TX_WR_PTR(s) + 0);
14   wr_ptr.cVal[2] = *(TX_WR_PTR(s) + 1);
15   wr_ptr.cVal[1] = *(TX_WR_PTR(s) + 2);
16   wr_ptr.cVal[0] = *(TX_WR_PTR(s) + 3);
17
18   k = *SHADOW_TXRD_PTR(s);
19   wait_1us(2);
20   rd_ptr.cVal[3] = *(TX_RD_PTR(s) + 0);
21   rd_ptr.cVal[2] = *(TX_RD_PTR(s) + 1);
22   rd_ptr.cVal[1] = *(TX_RD_PTR(s) + 2);
23   rd_ptr.cVal[0] = *(TX_RD_PTR(s) + 3);
24   sei();
25
26   if (wr_ptr.lVal >= rd_ptr.lVal)
27    size = (u_int) (SSIZE[s] - (wr_ptr.lVal - rd_ptr.lVal));
28   else
29    size = (u_int) (SSIZE[s] - (0 - rd_ptr.lVal + wr_ptr.lVal));
30
31   if (size > SSIZE[s]) { // Recalulate after some delay because
      of error in pointer caluation
32    wait_1ms(2);
33    goto S2_START;
34   }
35
36   if (size == 0) {
37    // Wait when previous sending has not finished yet and there
      's no free buffer
38   wait_1ms(2);
39   goto S2_START;
40   }
41   else {
42    if (size < len) len = size;}
43
44  send_ptr = (u_char*)(SBUFBASEADDRESS[s] + (UINT)(wr_ptr.lVal &
     SMASK[s]));
45   // Calculate pointer to copy data pointer
46
47  write_data(s, (u_char*) buf, send_ptr, len); // Copy data
48
49  if(select(s,SEL_CONTROL)==SOCK_CLOSED) return -1; // Error
50
51  wr_ptr.lVal = wr_ptr.lVal + len;  // Update tx_wr_ptr
52
53  *(TX_WR_PTR(s) + 0) = wr_ptr.cVal[3];
54  *(TX_WR_PTR(s) + 1) = wr_ptr.cVal[2];
55  *(TX_WR_PTR(s) + 2) = wr_ptr.cVal[1];
56  *(TX_WR_PTR(s) + 3) = wr_ptr.cVal[0];
57
58  COMMAND(s) = CSEND; // SEND
59  return (len);
60 }
```

5.4.3 u_int read_data(SOCKET s, u_char* src, u_char* dst, u_int len)

Die Funktion `write_data` kopiert soviele Daten von der Quelle *∗src* zum Ziel *∗dst*, wie in *len* angegeben. Falls das Ende der Daten über das Ende des Empfangspuffers hinaus geht, wird der Rest der Daten vom Anfang des Speicherbereichs kopiert.

Listing 40: `read_data`

```
1  u_int read_data(SOCKET s, u_char* src, u_char* dst, u_int len)
   {
2   u_int i, size, size1;
3
4   if (len == 0) return 0;
5
6   if( (((u_int)src & RMASK[s]) + len)  > RSIZE[s]) {
7    size = RSIZE[s] - ((u_int)src & RMASK[s]);
8    for (i = 0; i < size; i++)
9     *dst++ = *src++;
10   size1 = len - size;
11   src =  RBUFBASEADDRESS[s];
12   for (i = 0; i < size1; i++)
13    *dst++ = *src++;
14  }
15  else {
16   for (i = 0; i < len; i++)
17    *dst++ = *src++;
18  }
19  return len;
20 }
```

5.4.4 u_int write_data(SOCKET s, u_char* src, u_char* dst, u_int len)

Die Funktion `write_data` kopiert soviele Daten von der Quelle *∗src* zum Ziel *∗dst* wie in *len* angegeben. Falls das Ende der Daten über das Ende des Sendepuffers hinaus geht, wird der Rest der Daten an den Anfang des Speicherbereichs kopiert.

Listing 41: `write_data`

```
1  u_int write_data(SOCKET s, u_char* src, u_char* dst, u_int len)
    {
2   u_int i, size, size1;
3
4   if (len == 0) return 0;
5
6   if ( (((u_int)dst & SMASK[s]) + len) > SSIZE[s]) {
7    size = SSIZE[s] - ((UINT)dst & SMASK[s]);
8    for (i = 0; i < size; i++)
9     *dst++ = *src++;
10   size1 = len - size;
11   dst = (SBUFBASEADDRESS[s]);
12   for (i = 0; i < size1; i++)
```

```
13      *dst++ = *src++;
14    }
15    else {
16     for (i = 0; i < len; i++)
17      *dst++ = *src++;
18    }
19    return len;
20  }
```

5.5 Hilfsfunktionen

Delay-Funktionen

Um wenn nötig, dem Netzwerk-Chip Zeit zu geben, die Befehle und insbesondere Speicheroperationen auszuführen, werden Delay-Funktionen benutzt.

Listing 42: `wait_1µs`

```
void wait_1us(int cnt){
 cnt=cnt*8; // we are running at 7,5Mhz, so lets wait a little bit longer
 for (u_int i = 0; i < cnt; i++);
}
```

Listing 43: `wait_1ms`

```
void wait_1ms(int cnt){
 for (u_int i = 0; i < cnt; i++) wait_1us(1000);
}
```

Listing 44: `wait_10ms`

```
void wait_10ms(int cnt){
 for (u_int i = 0; i < cnt; i++) wait_1ms(10);
}
```

5.6 Big Endian ↔ Little Endian - Konvertierung

Da sich die Byte-Order des Atmel von der des w3100a unterscheidet, benötigt man eine Funktion die die Werte entsprechend konvertiert.

Listing 45: `swapuint`

```
u_int swapuint(u_int i){
 return ((i & 0x00FF) << 8) | ((i & 0xFF00) >> 8);
}
```

6 Fazit

Mit der Bibliothek die aus dieser Studienarbeit hervorging können schnell und portierbar Netzwerkapplikationen programmiert werden. Diese wird schon jetzt, im Rahmen eines Projektpraktikums welches sich mit der Steuerung eines Modellautos über eine WLAN-Verbindung, verwendet. Hierbei übernimmt der Mikrocontroller eine reine Gateway-Funktion, d.h. er übernimmt keine aktive Aufgaben in der Steuerung des Fahrzeugs, sondern ist nur für den Datentransfer zuständig. Es sind aber auch einfacherer Anwendungen denkbar, bei denen keine weiteren Mikrocontroller benötigt werden. Die möglichen Anwendungsszenarien reichen von einer übers Netzwerk einzuschaltenden Kaffeemaschine bis zu einer Wetterstation, deren Sensoren übers Netzwerk abgefragt werden können.

Anhang

Befehlsregister des w3100a

Address	Register	Bit Definitions							
0x00	C0_CR	S/W Reset	Recv	Send	Close	Listen	Connect	Sock_Init	Sys_Init
0x01	C1_CR	Memory Test	Recv	Send	Close	Listen	Connect	Sock_Init	
0x02	C2_CR		Recv	Send	Close	Listen	Connect	Sock_Init	
0x03	C3_CR		Recv	Send	Close	Listen	Connect	Sock_Init	
0x04	C0_ISR		Recv_OK	Send_OK	Timeout	Closed	Established	SInit_OK	Init_OK
0x05	C1_ISR		Recv_OK	Send_OK	Timeout	Closed	Established	SInit_OK	
0x06	C2_ISR		Recv_OK	Send_OK	Timeout	Closed	Established	SInit_OK	
0x07	C3_ISR		Recv_OK	Send_OK	Timeout	Closed	Established	SInit_OK	
0x08	IR	C3R	C2R	C1R	C0R	C3	C2	C1	C0
0x09	IMR	IM_C3R	IM_C2R	IM_C1R	IM_C0R	IM_C3	IM_C2	IM_C1	IM_C0

Tabelle 2: Befehlsregisterübersicht des w3100a

Write–, Read– und Ack–Pointer des w3100a

0x10 – 0x13	C0_RW_PR	Channel 0 Rx Write Pointer Register
0x14 – 0x17	C0_RR_PR	Channel 0 Rx Read Pointer Register
0x18 – 0x1B	C0_TA_PR	Channel 0 Tx ACK Pointer Register
0x1C – 0x1F	C1_RW_PR	Channel 1 Rx Write Pointer Register
0x20 – 0x23	C1_RR_PR	Channel 1 Rx Read Pointer Register
0x24 – 0x27	C1_TA_PR	Channel 1 Tx ACK Pointer Register
0x28 – 0x2B	C2_RW_PR	Channel 2 Rx Write Pointer Register
0x2C – 0x2F	C2_RR_PR	Channel 2 Rx Read Pointer Register
0x30 – 0x33	C2_TA_PR	Channel 2 Tx ACK Pointer Register
0x34 – 0x37	C3_RW_PR	Channel 3 Rx Write Pointer Register
0x38 – 0x3B	C3_RR_PR	Channel 3 Rx Read Pointer Register
0x3C – 0x3F	C3_TA_PR	Channel 3 Tx ACK Pointer Register
0x40 – 0x43	C0_TW_PR	Channel 0 Tx Write Pointer Register
0x44 – 0x47	C0_TR_PR	Channel 0 Tx Read Pointer Register
0x48 – 0x4B	Reserved	
0x4C – 0x4F	C1_TW_PR	Channel 1 Tx Write Pointer Register
0x50 – 0x53	C1_TR_PR	Channel 1 Tx Read Pointer Register
0x54 – 0x57	Reserved	
0x58 – 0x5B	C2_TW_PR	Channel 2 Tx Write Pointer Register
0x5C – 0x5F	C2_TR_PR	Channel 2 Tx Read Pointer Register
0x60 – 0x63	Reserved	
0x64 – 0x67	C3_TW_PR	Channel 3 Tx Write Pointer Register
0x68 – 0x6B	C3_TR_PR	Channel 3 Tx Read Pointer Register

Tabelle 3: Write–, Read– und Ack–Pointer des w3100a

Konfigurationsregister des w3100a

0x80 – 0x83	GAR	Gateway Address Register
0x84 – 0x87	SMR	Subnet Mask Register
0x88 – 0x8D	SHAR	Source Hardware Address Register
0x8E – 0x91	SIPR	Source IP Address Register
0x92 – 0x93	IRTR	Initial Retry Time-value Register
0x95	RMSR	Rx data Memory Size Register
0x96	TMSR	Tx data Memory Size Register

Tabelle 4: Konfigurationsregister des w3100a

Kanalspezifische Register des w3100a

0xA0	C0_SSR	Channel 0 Socket Status Register							
0xA1	C0_SOPR	Broadcast/ERR	NDTimeout/B	NDAck	SWS/P		Protocol	Protocol	Protocol
0xA2 – 0xA7	Reserved								
0xA8 – 0xAB	C0_DIR	Channel 0 Destination IP Address Register							
0xAC – 0xAD	C0_DPR	Channel 0 Destination Port Register							
0xAE – 0xAF	C0_SPR	Channel 0 Source Port Register							
0xB0	C0_IPR	Channel 0 IP Protocol Register							
0xB1	C0_TOSR	Channel 0 TOS (type of service) Register							
0xB2 – 0xB3	C0_MSSR	Channel 0 MSS (maximum segment size) Register							

Tabelle 5: kanalspezifische Register des w3100a (die anderen Kanäle analog 0x18 verschoben)

Auf der beiliegenden CD befinden sich folgenden Anhänge:

- Source-Code der Bibliothek
- Source-Code der Beispielprogramme
- Eagle-Dateien der Platinen
- dieses Dokument

Codeverzeichnis

Abbildungsverzeichnis

Literatur

[Inc] Inc, WIZnet: *IIM7000 Datasheet*. http://tinyurl.com/o4nph.

[Pos] Postel, J.: *ASSIGNED NUMBERS*. http://tools.ietf.org/html/rfc790.

[Ram] Ramakrishnan, et al.: *ASSIGNED NUMBERS*. http://tools.ietf.org/html/rfc3168.

[The] TheOpenGroup: *The Single UNIX Specification, Version 2*. http://opengroup.org/onlinepubs/007908799/xns/syssocket.h.html.

[WC] W3 Consortium, Fielding, et al.: *Hypertext Transfer Protocol*.

[Wika] Wikipedia: *Wikipedia Byteorder*. http://de.wikipedia.org/wiki/Byte-Reihenfolge.

[Wikb] Wikipedia: *Wikipedia POSIX*. http://de.wikipedia.org/wiki/POSIX.

[Wikc] Wikipedia: *Wikipedia RJ45*. http://de.wikipedia.org/wiki/RJ45.

[Wikd] Wikipedia: *Wikipedia TCP*. http://de.wikipedia.org/wiki/Transmission_Control_Protocol.

[Wike] Wikipedia: *Wikipedia UDP*. http://de.wikipedia.org/wiki/UDP.

[WIZ] WIZnet, Inc.: *WIZnet Products*. http://www.iinchip.com/wiznet/product_assp.html.