Projektarbeit

23. Mai 2000 - 21. Juli 2000

Mini Web Server mit SSL

Machbarkeitsstudie

A. Zingg
B.Lenzlinger

Inhaltsverzeichnis

1 Zusammenfassung

Seit jeher hat der Mensch ein spezielles Interesse an geheimen oder nicht für ihn bestimmten Informationen. Da drängt sich das Thema Sicherheit geradezu auf, denn welcher Geschäftsmann hat nicht das Bedürfnis, seine Firmendaten vor der Konkurrenz zu verbergen. Dies kann bei einer Übertragung über das Internet nur mit kryptographischen Methoden verwirklicht werden. Dies führte zu einem neuen Marktsegment: der Internet Security.

Der IPC@CHIP von Beck wurde für die Entwicklung von Industrieapplikationen designt. Mit der integrierten Ethernetschnittstelle eröffnen sich neue Dimensionen im Bezug auf die Fernwartung und Datenübertragung vià Internet. Doch der IPC@CHIP unterstützt leider keinerlei Sicherheitsmechanismen.

Ziel unserer Projektarbeit war es herauzufinden, ob sich ein Verschlüsselungsmechanismus auf der Basis einer IPC@CHIP-Applikation realisieren lässt.

Der Lösungsansatz heisst SSL (Secure Sockets Layer). SSL implementiert diverse Verschlüsselungsalgorithmen sowie die geheime Generierung eines Session Keys und gewährleistet somit einen vor Fremdzugriff geschützten Datenfluss. SSL ist als Open Source Software frei zugänglich (OpenSSL) und beliebig erweiterbar.

Unsere Untersuchungen ergaben, dass OpenSSL mit gewissen Einschränkungen auf die IPC@CHIP-Plattform portiert werden kann. Die begrenzten Ressourcen erlauben jedoch nur eine minimale Implementation von OpenSSL. Bei einer konkreten Portierung muss mit Rücksicht auf eben diese Ressourcen allenfalls der Zertifikatsaustausch weggelassen werden, was OpenSSL auf ein einfacheres Challenge/Response-Protokoll reduzieren würde. Dieses ist nicht weniger sicher, aber es wäre dann unumgänglich, einen Client mit den nötigen Sicherheitsaspekten zu implementieren. Könnte OpenSSL hingegen ganz portiert werden, ist ein SSL-fähiger Internet-Browser (Netscape, Internet Explorer) als Client prädestiniert.

Diese Entscheidung konnten wir in der beschränkten Zeit unserer Projektarbeit nicht treffen. Sie wird aber in einer kommenden Diplomarbeit viel Diskussionsstoff und Denkarbeit liefern.

Winterthur, 21. Juli 2000

Bernhard Lenzlinger

Andreas Zingg

2 Aufgabenstellung

Kommunikationssysteme (KSy)

Projektarbeiten SS 2000 - Sna06

Mini-Webserver mit SSL

Studierende:

- Bernhard Lenzlinger, IT3b
- Andreas Zingg, IT3b

Termine:

- Ausgabe: Dienstag, 23.05.2000 11:00 - 12:00 im E509
- Abgabe: Freitag, 21.07.2000

Beschreibung:

Mit dem IPC@CHIP von Beck lässt sich für wenig Geld ein Web-Server mit integrierter Ethernet-Schnittstelle realisieren, der über 24 V Ein- und Ausgänge direkt Hardwareschnittstellen bedienen kann. Damit eröffnet sich die Möglichkeit, über das Internet weltweit Geräte ansteuern zu können. Als Konsequenz dieser globalen Vernetzung rücken natürlich Sicherheitsaspekte in den Vordergrund. Der Zugriff auf diese Geräte sollte nur Berechtigten vorbehalten werden. Weil offen über das Internet gesendete Passwörter leicht abgehört werden können, sollte ein Challenge/Response Protokoll für die Authentisierung verwendet werden. Zusätzlich sollte der gesamte Datenaustausch verschlüsselt werden können. Als Standardlösung bietet sich zu diesem Zweck der Secure Sockets Layer (SSL) an, der es erlaubt, mit dem Web-Server sicher zu kommunizieren.

Im Rahmen dieser Projektarbeit soll abgeklärt werden, ob die als C-Source Code verfügbare OpenSSL-Library auf den IPC@CHIP portiert werden kann. Insbesondere muss die Anbindung an das TCP-IP API des IPCs untersucht werden.

Aufgaben:

- Inbetriebnahme des IPC@CHIPs im LAN
- Aufsetzen der C-Entwicklungsumgebung für 16 Bit Prozessoren, z. Bsp.
  - Microsoft Visual C/C++ V1.52
  - Microsoft C V6.00
  - Borland Turbo C/C++ V1.00 - V3.00,
  - Borland C/C++ V5.02
- Studium des IPC@CHIP TCP-IP APIs
- Realisierung einer einfachen TCP/IP Anwendung, z.Bsp. ein Mini-HTTP-Server
- Studium der SSL 3.0 Spezifikation
- Analyse des OpenSSL Stacks und Abklärung der Portierbarkeit
- Falls Die Zeit ausreicht und die Machbarkeit gewährleistet ist, Beginn der Portierung.

Infrastruktur / Tools:

- Raum: **E416**
- Rechner: 2 PCs, 2 SC12 IPC@CHIP Prozessoren mit DK40 Development Kit
- SW-Tools: C/C++ Compiler für 80186 Prozessoren, OpenSSL C-Library

Literatur / Links:

- Beck IPC@CHIP Seite
  http://www.beck-ipc.com/chip/
- OpenSSL Project
  http://www.openssl.org
- IETF Draft <draft-freier-ssl-version3-02.txt>
  The SSL Protocol Version 3.0
  - SSL 3.0 Implementation Assistance
    http://home.netscape.com/eng/ssl3/traces/index.html
  - Borland Museum
    http://community.borland.com/museum/

21.5.2000 by Andreas Steffen

3 Einleitung

Dieses Dokument ist folgendermassen aufgebaut:

- **Kapitel 3: Einleitung**
  Übersicht über die zur Verfügung stehende Ausrüstung, über den IPC@CHIP und über SSL.
- **Kapitel 4: Projektplan**
  Zeitliche Planung der Projektarbeit.
- **Kapitel 5: Vorgehen**
  Vorgehen während der Projektarbeit.
- **Kapitel 6: Inbetriebnahme IPC@CHIP**
  Technische Details zur Inbetriebnahme des IPC@CHIP.
- **Kapitel 7: OpenSSL**
  Installation von OpenSSL und Beschreibung der Demo-Programme.
- **Kapitel 8: Implementation**
  Der Weg zu OpenSSL für den IPC@CHIP.
- **Kapitel 9: Ausblick**
  Wie geht es weiter?
- **Kapitel 10: Schlusswort**
  Persönliche Erfahrungen während dieser Projektarbeit.
- **Anhang**
  Konfigurationen, Literathurhinweise, Links und Source-Code.

3.1Infrastruktur

3.1.1 Hardware

PC's: 2 Windows NT, 1 SuSE Linux 6.2
IPC@CHIP SC12 (Chip)
IPC@CHIP DK40 (Evaluation Kit)
IPC PS1 SM14 (Programmierkabel)
Netzgerät (Inv.Nr. HF 77.3)

3.1.2 Software

IP-Adresse für IPC@CHIP (160.85.134.67)
ChipTool, TeraTerm Pro, WS-FTP
OpenSSL 0.9.5a
(auf der CD vorhanden: `[CDROM]\OpenSSL\openssl-0.9.5a.tar.gz`)
Borland C++ 3.0 Compiler

3.2Übersicht IPC@CHIP

"Man nehme einen 186er-CPU-Kern (20MHz), je 512 KByte Flash-Speicher und RAM, ein Ethernet Interface, weitere Systemschnittstellen und etwas Glue Logic, setze das ganze auf eine Trägerplatine, vergiesse es in ein DIL-40-Gehäuse und nenne es IPC@CHIP SC12. Fertig ist der netzwerkfähige Industrie-PC, der sich als Modul in eigene Schaltungen einsetzen lässt."

c't 2000, Heft 2, Seite 84

Bei den Industrie-PCs (IPC) gibt es heute zwei Tendenzen zu beobachten. Einerseits entwickelt sich der IPC immer mehr zu einem komplexen System, dessen Ziel es ist, alles mit einem Prozessor erledigen: Steuern, Regeln, Positionieren, Visualisieren und Daten speichern. Die andere Entwicklungsrichtung geht zum immer kleiner werdenden IPC. Dabei wird bewusst auf einige PC-typische Features verzichtet. Weder Monitor- noch Tastaturanschluss sind zu finden. Das BIOS ist so aufgebaut, dass die erste serielle Schnittstelle als Terminalanschluss fungiert. Es ist also ein externes Entwicklungssystem nötig, um Programme zu entwickeln und in den IPC zu laden.

Der IPC@CHIP SC 12 der Firma Beck verwirklicht eine ebenso einfache wie komplizierte Idee: Alle Komponenten eines Industrie-PC auf einem CHIP vereinen.

- Prozessor-Kern.
- Arbeitsspeicher.
- Programmspeicher.
- Serielle Schnittstelle.
- Ethernet Schnittstelle fürs Netzwerk.
- Daten- und Adressbus für Ein-/Ausgänge.

Ursprünglich für kleine Industrieanwendungen geplant und entwickelt, wurde schnell klar, dass sich der in Hochsprachen programmierbare IPC auch für DOS- und Internetanwendungen bestens eignet.

Die integrierten APIs (TCP-IP API, DOS API,..) erlauben, kleinere Programme für fast alle Internetanwendungen (Web-Server, Telnet, FTP, ..) zu entwickeln.

Weitere Informationen zu dem IPC@CHIP finden Sie auf der CD im Verzeichnis `[CDROM]\Beck\Documentation` oder auf der Website von Beck [3].

3.3Übersicht SSL

Die Internetarchitektur beruht auf verschiedenen Schichten von Protokollen, von den jedes auf den Services des darunterliegenden aufbaut. Viele dieser unterschiedlichen Protokollschichten können Sicherheitsdienstleistungen unterstützen, und alle haben ihre eigenen Vor- und Nachteile. Während für SSL eine neue Protokollschicht eingeführt wurde, gibt es auch noch die Möglichkeit, Security Services direkt in das Application Protocol oder in ein Core Network Protocol (Bsp. IP) zu integrieren. Als eine weitere Alternative können die Security Services auch in einem sog. Parallel Protocol im Application Layer enthalten sein.

In Tabelle 3.1 sind die unterschiedlichen Methoden zum Erreichen der Netzsicherheit mit ihren Vor- und Nachteilen aufgelistet:

Protocol Architecture	Example	A	B	C	D	E
Separate Protocol Layer	SSL	●	●	○	○	●
Application Layer	S-HTTP	●	○	●	○	●
Integrated with Core	IPSEC	●	●	○	●	○
Parallel Protocol	Kerberos	○	●	○	○	●
Benefits: A – Full Security B - Multiple Applications C - Tailored Services D – Transparent to Application E - Easy to Deploy						

Tabelle 3.1: Verschiedene Methoden zum Erreichen der Netzsicherheit (aus [1])

Das SSL-Protokoll soll eine sichere Kommunikation über das Internet ermöglichen. Es setzt sich hauptsächlich aus zwei Schichten zusammen. Die untere Schicht wird durch den SSL Record Layer, die obere durch das SSL Handshake Protocol gebildet. Der SSL Record Layer setzt auf einem zuverlässigen Transportprotokoll auf (Bsp. TCP) und dient der Kapselung von höheren Protokollen. Das SSL Handshake Protocol liegt über dem SSL Record Layer und ermöglicht das Aushandeln von Verschlüsselungsalgorithmen und zugehörigen Schlüsseln sowie die Authentifizierung des Servers oder des Clients. Dies geschieht, bevor die Übertragung zwischen höheren Schichten einsetzt. SSL ist für die angrenzenden Protokollschichten transparent und arbeitet unabhängig von den Protokollen der Anwendungsschicht.

Bild 3.1 zeigt, wie SSL in die Internet Protokollarchitektur integriert wurde.

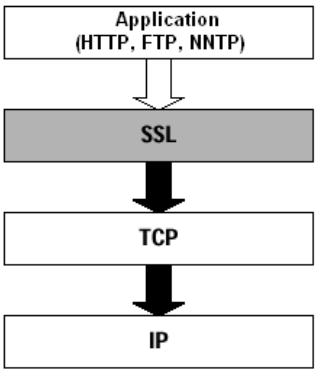

Bild 3.1: SSL im Schichtenmodell

Die Sicherheit in SSL wird durch folgende Punkte gewährleistet:

- Die Verbindung ist *privat*: Mittels Handshake wird ein *geheimer Schlüssel* ausgehandelt, der dann für die Übertragung aller Nachrichten verwendet wird.
- Eine *Authentifizierung* der Verbindungspartner ist möglich.
- Es findet eine *Integritätsprüfung* der versendeten Nachrichten mittels Message Authentication Code (MAC) statt.

Unter einem Message Authentication Code versteht man eine Einweg-Hashfunktion, die in Verbindung mit einem geheimen Schlüssel eingesetzt wird (Beispiel MD5, SHA).

Während einer sicheren Verbindung kommunizieren die beteiligten Rechner ausschliesslich über die Mechanismen, die von SSL bereitgestellt werden. Steht die sichere Verbindung nicht zur Verfügung, schaltet sich das SSL-Protokoll aus.

3.3.1 Handshake Protocol

In Bild 3.2 ist ein SSL-Verbindungsaufbau dargestellt.

1. Der Client sendet die **ClientHello** Nachricht. Diese beinhaltet unter anderem die Protokollversionsnummer, eine Session-ID (falls der Client eine alte Verbindung mit dieser Session-ID wiederaufzunehmen wünscht), eine Auswahl von kryptographischen Verfahren und Hashfunktionen (*cipher suites*) sowie Kompressionsverfahren, die der Client unterstützt.
2. Der Server antwortet entweder mit der **ServerHello** Nachricht oder einer Fehlermeldung und dem Abbruch der Verbindung. Ist keine Session-ID übermittelt worden, bestimmt der Server eine neue Session-ID, welche die neue Verbindung eindeutig kennzeichnet. Dann wählt er eine der angebotenen cipher suites sowie ein Kompressionsverfahren aus und teilt diese dem Client mit.
3. Anschliessend sendet der Server mit der **Certificate** Nachricht sein Zertifikat oder, wenn er kein entsprechendes besitzt, eine **ServerKeyExchange** Nachricht, die nur für den Schlüsselaustausch gedacht ist. Im Folgenden wird davon ausgegangen, dass der Server ein gültiges Zertifikat besitzt. Im Allgemeinen handelt es sich dabei um ein X.509 v3 Zertifikat. Wenn sich auch der Client gegenüber dem Server authentifizieren soll, sendet der Server eine **CertificateRequest** Nachricht. Zum Abschluss folgt eine **ServerHelloDone** Nachricht, die das Ende der ServerHello Nachricht und anschliessender Nachrichten anzeigt.

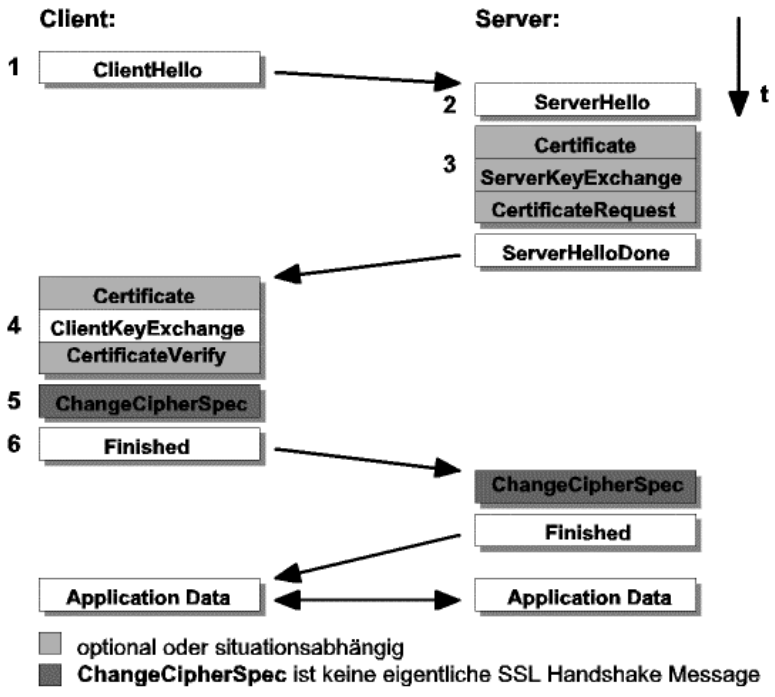

Bild 3.2: SSL-Verbindungsaufbau

4. Nach Empfang der ServerHelloDone Nachricht sendet der Client, wenn es vom Server gefordert wird, mit **Certificate** sein Zertifikat. In Abhängigkeit von dem vom Server ausgewählten Public-Key-Verfahren sendet der Client eine **ClientKeyExchange** Nachricht. Die ClientKeyExchange Nachricht entählt das vom Client erzeugte sog. *premaster secret*.
   Wird die Authentifizierung des Clients gefordert, sendet der Client die **CertificateVerify** Nachricht. Erst jetzt erfolgt die tatsächliche Authentifizierung des Clients. Denn anhand dieser Nachricht lässt sich vom Server überprüfen, ob der Client auch im Besitzt des zum Zertifikat gehörenden privaten Schlüssels ist.

   Client als auch Server generieren aus dem vom Client erzeugten *premaster sectret* eine für Client und Server gemeinsame, geheime Information, das sog. *master secret*. Aus diesem *master secret* werden unter anderem die Schlüssel für die Verschlüsselungsalgorithmen und MAC-Berechnungen erstellt.

5. Dann sendet der Client eine **ChangeCipherSpec** Nachricht, die mit den vereinbarten Verfahren verschlüsselt und komprimiert wird. Sie ist nicht eigentlicher Bestandteil des SSL Handshake Protocol. Die ChangeCipherSpec Nachricht soll anzeigen, dass die folgenden Nachrichtenblöcke mit den vereinbarten Verfahren bearbeitet werden.
   Der Server sendet seine **ChangeCipherSpec** Nachricht, nachdem er die ClientKeyExchange Nachricht erfolgreich verarbeitet und die nötigen Schlüssel generiert hat. Eine unerwartete ChangeCipherSpec Nachricht führt immer zu einer Fehlermeldung und zum Abbruch der Verbindung.
6. Den Abschluss des Handshakes bilden die **Finished** Nachrichten. Sie werden sofort nach den ChangeCipherSpec Nachrichten gesendet und sollen den erfolgreichen Schlüsselaustausch und die erfolgreiche Authentifizierung verifizieren. Nach der Finished Nachricht können bereits vertrauliche Daten versendet werden, ohne dass auf eine Bestätigung gewartet werden muss.

Wiederaufnahme einer alten Verbindung

Die Möglichkeit der Wiederaufnahme alter SSL-Verbindungen soll die Zeit für den Handshake verkürzen. In Bild 3.3 ist der Mechanismus gezeigt, der abläuft, wenn der Client eine alte Verbindung wiederaufnehmen möchte.

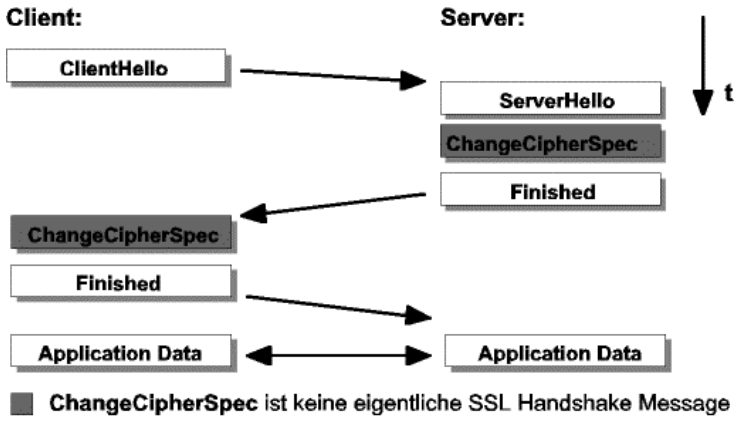

Bild 3.3: Wiederaufnahme einer alten Verbindung

Der Client übermittelt in der ClientHello Nachricht die Session-ID der wiederaufzunehmenden Verbindung. Findet der Server diese spezielle Session-ID in

seinem Cache und möchte er die die Verbindung wieder unter den alten Vereinbarungen aufsetzen, sendet er seine ServerHello Nachricht mit der gleichen Session-ID. Nun müssen nur noch die ChangeCipherSpec Nachrichten gefolgt von den Finished Nachrichten gesendet werden. Danach kann der vertrauliche Datenaustausch erfolgen.

Findet der Server geforderte Session-ID in seinem Cache nicht, sendet er die ServerHello Nachricht mit einer neuen Session-ID zum Client zurück und der gesamte Handshake muss durchgeführt werden.

Für detailiertere Informationen sei auf das Buch *SSL and TLS Essentials* [1] und auf die Webseite von Netscape [6] verwiesen.

4 Projektplan

Es fiel uns zu Beginn der Projektarbeit recht schwer, einen Zeitplan aufzustellen, da wir uns nur sehr schwer vorstellen konnten, was genau an zeitlichem Aufwand auf uns zu kommen würde. Daher haben wir die Zeiten relativ grosszügig verteilt.

Im Vergleich von geschätztem und effektivem Zeitplan geht hervor, dass wir einige Arbeiten unterschätzt hatten. Die Inbetriebnahme des IPC@CHIP dauerte länger als erwartet, wie auch die Einrichtung der Entwicklungsumgebung für den 80186-Prozessor. Bei den restlichen Aufgaben waren wir im gesetzten Rahmen oder knapp darüber, weshalb die Zeit für die Portierung nicht mehr ausreichte.

Die genauen Abläufe und Überlegungen sind in Kapitel 5 dokumentiert.

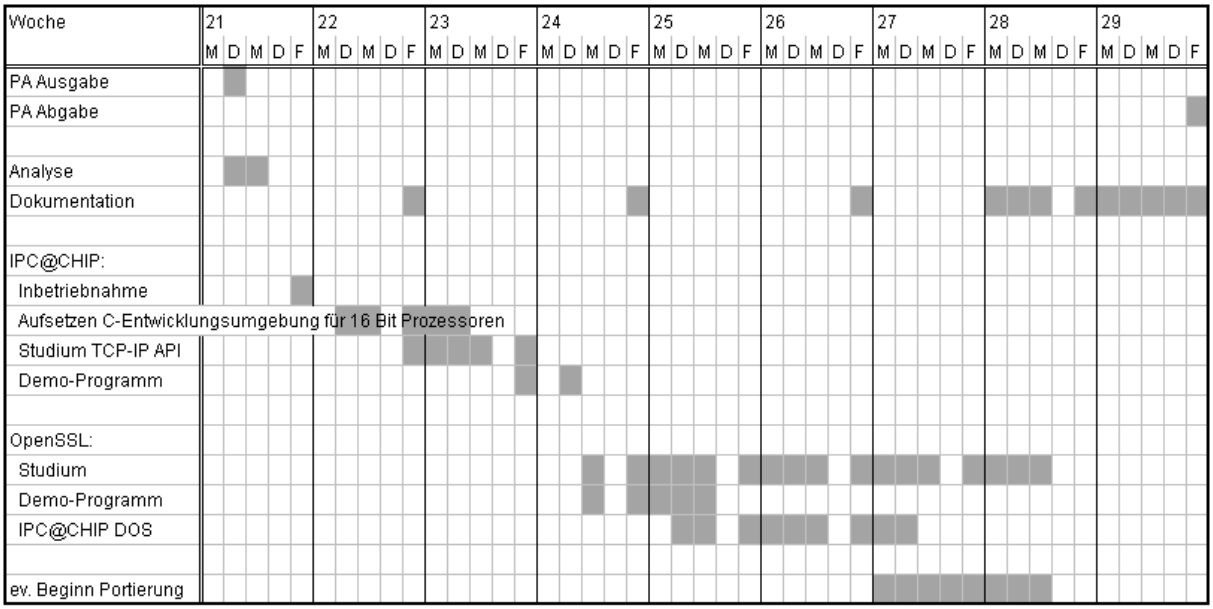

Bild 4.1: Projektplan mit den geplanten Zeiten.

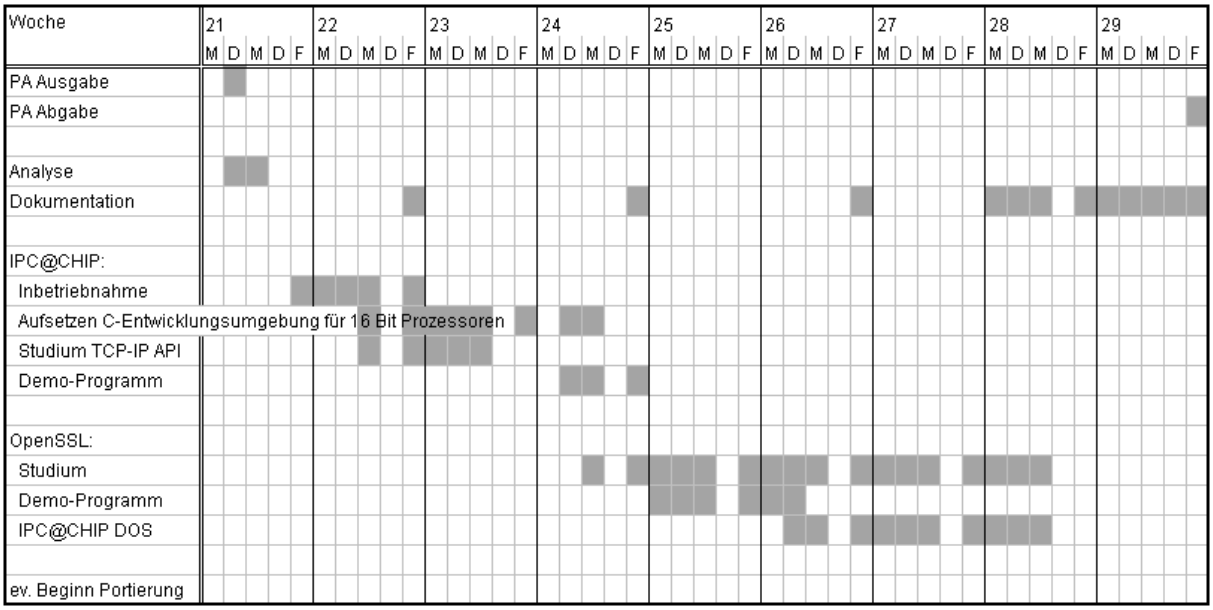

Bild 4.2: Projektplan mit den effektiven Zeiten.

5 Vorgehen

In diesem Kapitel werden die verschiedenen Aufgaben aus Aufgabenstellung und Projektplan und unsere Lösungen bzw. Lösungsansätze genauer dokumentiert.

Zuerst ein Grobüberblick: Zuerst wollten wir den IPC@CHIP in Betrieb nehmen, dann OpenSSL unter Linux installieren. Wir nahmen uns vor unter Linux arbeiten, da OpenSSL aus diesem Lager stammt, und die Entwicklung eines Demoprogrammes viel einfacher vonstatten gehen würde.

Sobald OpenSSL funktioniert, wollten wir es für den IPC@CHIP anpassen.

5.1IPC@CHIP

5.1.1 Inbetriebnahme

Die Inbetriebnahme des IPC@CHIP gestaltete sich sehr einfach, da die Anleitung von Beck die genaue Abfolge beschreibt. Unser grösstes Problem bestand darin, dass wir einen defekten Chip erhalten hatten. Hr. Steffen gab uns seinen Chip, doch auch dieser hatte ein Problem. Bei der Initialisierung des Watch Dog wurde ein erneuter Reset ausgelöst, was eine Endlosschlaufe zur Folge hatte. Wir versuchten, die aktuelle BIOS-Version auf dem Chip (v0.66) durch v0.65 zu ersetzen, was auch gelang. Der Chip startete jetzt einwandfrei. Nun machten wir wieder den Update von v0.65 zu v0.66. Der Chip startete immer noch einwandfrei, die Inbetriebnahme war geglückt.

5.1.2 Aufsetzen der Entwicklungsumgebung

Die Entwicklungsumgebung warf uns in die guten alten Zeiten von DOS zurück. Da das Herz des IPC@CHIP aus einer 80186 CPU besteht, der unter einem Betriebssystem ähnlich zu DOS 3.0 läuft (abgespeckte Version, aber mit einem TCP-Stack), mussten wir eine 16-Bit-Entwicklungsumgebung finden. Wir entschieden uns, mangels Alternativen, Borland C++ 3.0 für DOS (im folgenden BC genannt) einzusetzen. Die Installation dieser Software unter Windows NT klappte einwandfrei. Als nächstes mussten wir ein Demoprogramm von Beck kompilieren, linken und auf dem IPC@CHIP laufen lassen; erst dann würde klar sein, ob sich BC zur Entwicklung eignet (siehe 5.1.3).

5.1.3 Demo-Programm

Da auf der Homepage von Beck [3] diverse Demoprogramme zum Download bereitstehen, haben wir uns nicht die Mühe gemacht, ein eigenes Programm zu schreiben, das die TCP-Funktionalität des IPC@CHIP demonstriert. Vielmehr haben wir einen Echo-Server und einen TCP-Client heruntergeladen (Server: *tcpserv.exe*; Client: *tcpclnt.exe*, muss installiert werden aus tcpclnt.zip). Die Programme sind auf der CD vorhanden (siehe Anhang B:).

Beim Server war nur der Sourcecode vorhanden. Wir kompilierten denselben mit BC und kopierten die ausführbare Datei per FTP auf den IPC@CHIP. Dort gestartet horchte der Server auf dem Port 7 und schickte die Daten vom Client eins zu eins zurück. Der Client zeigte dieselben Daten an, was bedeutete, dass sowohl BC als Entwicklungsumgebung, als auch die TCP-Funktionalität des IPC@CHIP einwandfrei funktionierten.

5.1.4 TCP-IP API

Die Änderungen und Ergänzungen am TCP-IP API des IPC@CHIP werden in Kapitel 5.2.3 beschrieben.

5.2OpenSSL

5.2.1 Installation

Die Installation von OpenSSL unter Linux verlief ohne Probleme. Sie ging auch sehr schnell vonstatten, so dass wir nach knapp einer Stunde eine funktionstüchtige Installation hatten.

Die genauen Schritte der Installation sind in Kapitel 7.4 beschrieben.

5.2.2 Demo-Programm

Wir behandeln hier nur die Probleme, die wir mit dem Demoprogramm hatten. Für eine Funktionsbeschreibung sei auf Kapitel 7.5 verwiesen.

Das Client- und das Serverprogramm werden mit OpenSSL mitgeliefert als *.cpp Dateien. Nach der Installation von OpenSSL wollten wir natürlich sofort die Demoprogramme unter die Lupe nehmen. Doch beim Kompilierungsversuch hagelte es Fehlermeldungen. Das lag daran, dass der Kompiler (*gcc*) die gewünschten Bibliotheken nicht finden konnte. Wir mussten sie in der Kommandozeile explizit angeben (Details siehe Kapitel 7.4). Auch die Umbenennung der *.cpp Dateien in *.c brachte einige Fehlermeldungen zum Verschwinden. Weiter mussten noch diverse Headerdateien hinzugefügt werden. Das Ganze vermittelte den Eindruck, als ob da ein Bastler am Werke gewesen war und nicht die OpenSSL-Projektgruppe.

Als wir die Programme endlich Kompilieren und Linken konnten, kamen die nächsten Herausforderungen auf uns zu. Die Programme stürzten ab. Nach vielen Stunden Arbeit fanden wir heraus, dass wir erst eigene Zertifikate generieren mussten (je ein Server- und ein Clientzertifikat mit korrespondierendem privatem Schlüssel). Nach der Generierung derjenigen (Details sind beschrieben in [5] ab Seite 18) funktionierte alles wie erwartet. Die SSL-Verbindung wurde aufgebaut und die Daten verschlüsselt übermittelt.

Wir passten das Server-Programm so an, dass mit dem Netscape-Browser darauf zugegriffen werden konnte und dieser eine kleine HTML-Seite erhielt. Doch auch hier hatten wir Probleme. Manchmal konnte die Verbindung nicht hergestellt werden. Laut den Statusmeldungen des Servers war es ein SSL-Versionskonflikt, aber die Fehler waren nicht reproduzierbar. An einem Tag konnte sich der Browser verbinden, am nächsten Tag nicht mehr, obwohl alle Einstellungen dieselben waren und der Server nicht neu übersetzt wurde.

Wenn jemand unsere Demoprogramme auf der CD testen will und diese nicht funktionieren, ist dies ein zu erwartender Fehler. Bis zur Druckzeit dieses Dokumentes waren wir leider nicht in der Lage, diesen Fehler zu beheben.

5.2.3 TCP-IP API des IPC@CHIP

OpenSSL benutzt für TCP Verbindung Standardfunktionen, die in einer Standard-Socketlibrary implementiert sind. Die Funktionen der TCP-IP API des IPC@CHIP werden jedoch mit Interrupts angesprochen, kennt also keine dieser Standardfunktionsnamen. Um die Softwarelücke zwischen diesen beiden

Komponenten zu schliessen, erweiterten wir die TCP-IP API des IPC@CHIP um ein Interface, das sich nach aussen nun wie eine (abgespeckte) Standard-Socketlibrary präsentiert, intern aber die IPC@CHIP-Interrupts des Betriebssystems aufruft (siehe auch Kapitel 8.1).

Das C- und das Headerfile sind auf der CD enthalten (siehe Anhang B:).

5.2.4 Makefile

Bei der Konfiguration von OpenSSL wird ein Makefile für das Zielsystem erstellt. Das generierte Makefile für Borland C war für BC 4.5 gedacht, nicht für BC 3.0, das wir verwendeten. Aus diesem Grund musste das Makefile von Hand an unsere Bedürfnisse angepasst werden bzw. an diejenigen des Dienstprogrammes *make*. Als Referenz diente das vom Perlskript generierte Makefile.

Eine detaillierte Beschreibung zum Makefile steht in Kapitel 8.2.1, das Makefile selber ist auf der CD enthalten (siehe Anhang B:).

5.2.5 OpenSSL für den IPC@CHIP

Nachdem das Makefile angepasst war, mussten wir theoretisch nur noch einen Build machen. Doch in der Praxis war vieles anders. Eine Hürde waren die langen Dateinamen unter Linux, die sich aber mit dem 8-3-Format von DOS nicht vertragen. So mussten wir mühsam diverse Header- und C-Dateien durchforsten und die Dateinamen anpassen.

Weiter ist der Datentyp `int` unter Linux 4 Byte lang, unter DOS aber nur 2 Byte. Das ergab unzählige Warnings. Das hatte zwar keinen Einfluss auf die Generierung der Binärdateien, aber natürlich auf deren Funktionseigenschaften. Im Rahmen dieser Projektarbeit reichte aber die Zeit nicht aus, die beschriebenen Warnings (und etliche mehr) zu beheben. Dies müsste in einer anschliessenden Diplomarbeit geschehen.

Das Ergebnis und die Schlüsse daraus sind im Ausblick in Kapitel 9 zusammengefasst.

6 Inbetriebnahme IPC@CHIP

Das Verfahren zur Inbetriebnahme ist detailliert beschrieben in der Datei `\Beck\Documentation\Inbetriebnahme\startup.pdf` auf der CD (siehe Anhang B:). Beim Anlegen der Betriebsspannung passierte ausser dem kurzen Aufleuchten der Kontroll-LED gar nichts, obwohl diese gemäss der Anleitung nach einer bestimmten Sequenz aufleuchten sollte. Nach dem erfolglosen Wechsel des DK40 war klar, dass der Chip defekt sein musste. Doch auch der neue Chip hatte einen Fehler. Beim Aufstarten ging alles gut bis zum Initialisieren des Watch Dog. Hier wurde immer wieder ein neuer Reset ausgelöst. Nach einem BIOS-Update auf die ältere Version 0.65 (die aktuelle Version auf dem Chip war V 0.66) funktionierte alles wie es sollte. Ein erneuter Update auf V 0.66 brachte den gewünschten Erfolg: der IPC@CHIP startete wie beschrieben.

6.1 Konfiguration

Nach einer Testphase mit fester IP-Adresse an einem nicht mit dem Schulnetz verbundenen LAN wurde der IPC@CHIP als DHCP Client konfiguriert und direkt ans Schulnetz angeschlossen. Dazu musste vorgängig eine IP-Adresse vom ZID beantragt werden. Der Miniwebserver war von diesem Zeitpunkt an unter *mini.zhwin.ch* oder *160.85.134.67* ansprechbar.

SC12 Seriennummer	`0016F`
MAC Adresse	`00 30 56 F0 01 6F`
IP Adresse	`160.85.134.67`
Subnetzmaske	`255.255.240.0`
Gateway	`160.85.128.1`

Tabelle 6.1: Konfiguration des IPC@CHIP

6.2 Tools

Alle im folgenden genannten Tools sind auf der CD enthalten (siehe Anhang B:).

- **TeraTerm Pro**
  TeraTerm Pro wurde als Terminalprogramm verwendet. Die korrekten Einstellungen für die Kommunikation mit dem SC12 sind folgende:

Baudrate	`19200`
Datenbits	`8`
Parität	`keine`
Stopbits	`1`
Fusskontrolle	`keine`

Tabelle 6.2: Korrekte Einstellungen für TeraTerm Pro

- **WS-FTP Lite**
  WS-FTP Lite wurde verwendet, um Daten vom und zum SC12 zu transferieren. Die genauen Login-Informationen sind in Anhang A: beschrieben.

- **ChipTool**
  Mit Hilfe des Programms ChipTool der Firma Beck kann der IPC@CHIP konfiguriert werden. Weiter können alle aktiven Chips in einem LAN gesucht und deren Statusinformationen angezeigt werden.

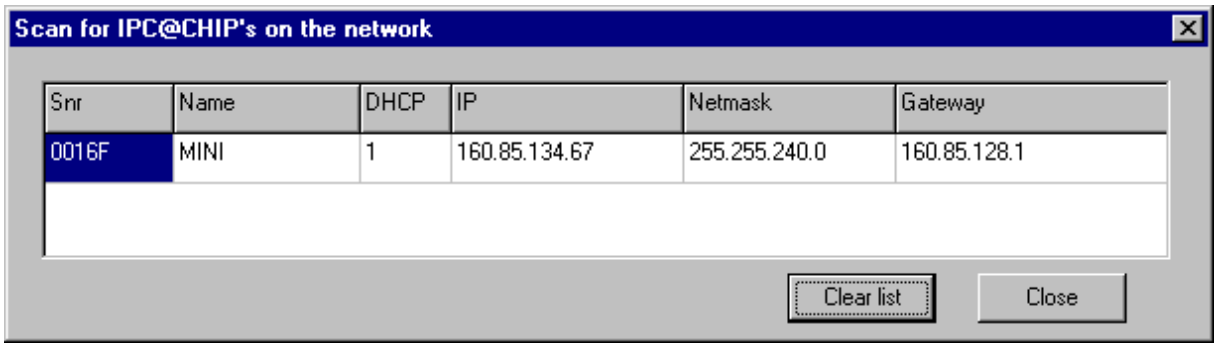

Bild 6.1: Statusinformationen aller im aktuellen LAN gefundenen IPC@CHIP

Auch das BIOS-Update wird mit diesem Programm durchgeführt. Dazu muss das neue BIOS als Hexfile vorliegen (auf der CD vorhanden, diehe Anhang B:).

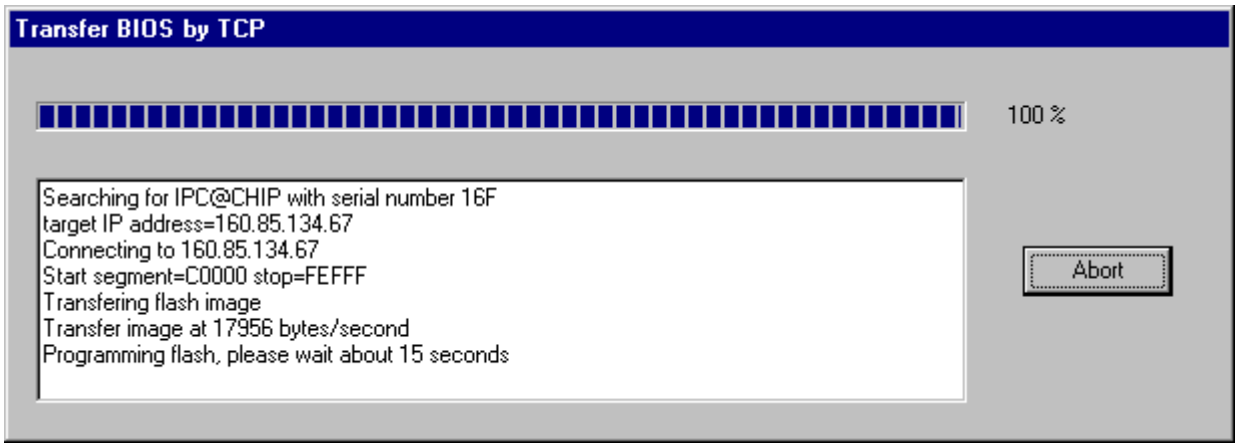

Bild 6.2: BIOS-Update auf V 0.66

6.3Eigene Homepage

Zum Test der Funktionalität des Miniwebservers wurde eine einfache Homepage auf dem IPC@CHIP installiert. Dazu mussten die HTML-Dateien per FTP in den Flashspeicher geladen und in der Ini-Datei (*chip.ini*) im Abschnitt *WEB* die Option *ROOTDIR* entsprechend angepasst werden.

```
[WEB]
ENABLE=1
ROOTDIR=WEB
```

6.4Demo Programm

Auf der Homepage von Beck [3] werden diverse Demoprogramme zum Herunterladen angeboten, zum Teil mit dem Sourcecode.

Das Programm *tcpclnt.exe* (muss installiert werden, *tcpclnt.zip*) baut eine TCP-Verbindung zu einem Echo-Server auf (*tcpserv.exe*), der auf dem IPC@CHIP (*mini.zhwin.ch*) läuft und auf Port 7 horcht.

Die Dateien sind auf der CD enhalten (siehe Anhang B:)

7 OpenSSL

7.1 Übersicht

OpenSSL ist ein Toolkit, der den Secure Sockets Layer (SSL v2/v3) sowie eine allgemeine Kryptographie-Bibliothek implementiert. Wie der Name schon sagt ist OpenSSL Open Source Software und somit frei verfügbar. Dieses Softwarepaket wird nicht einfach mit einem Install-Programm installiert, sondern muss auf jedem Zielsystem von Neuem kompiliert und gelinkt werden. Ein C-Compiler mit den dazugehörenden Dienstprogrammen (Make, Linker, Librarian) und die Skriptsprache Perl sind dabei Voraussetzung.

OpenSSL besteht zum einen aus den Header- und den Library-Dateien, zum anderen aus einem ausführbaren Programm. Mit diesem Programm können Zertifikate generiert, unterschrieben und zurückgezogen werden. Mit Hilfe der Header- und Library-Dateien kann SSL-Funktionalität in eigene Programme eingebaut werden.

Da OpenSSL eine freie Software ist, ist auch die Dokumentation sehr spärlich. Eine ausführliche deutsche Bedienungsanleitung zum Programm openssl ist auf dem Internet zu finden [5]

Der Toolkit und die Dokumentation sind erhältlich auf der Homepage von OpenSSL [4]

7.2 Hilfsmittel

Die Voraussetzung für die Installation von OpenSSL sind ein C-Compiler und die Skriptsprache Perl. Ohne diese kann OpenSSL nicht konfiguriert werden. Unter Linux ist Perl standardmässig installiert, für Windows NT muss dies nachträglich gemacht werden:

1. Ausführen von `[CDROM]\ActivePerl\nt\InstMsi.exe`
2. Installieren von Perl durch Ausführen von
   `[CDROM]\ActivePerl\ActivePerl-5_6_0_613.msi`

7.3 Installation unter Windows NT

Die Installation von OpenSSL unter Windows NT ist flexibler als unter Linux, da sie nicht in ein bestimmtes Verzeichnis erfolgen muss. Perl sowie Microsoft Visual C++ 6.0 müssen installiert sein. Die Pfadangaben können je nach Installation variieren.

Nachdem die Datei `[CDROM]\OpenSSL\openssl-0.9.5a.tar.gz` mit WinZip ins Verzeichnis `D:\openssl` entpackt wurde, kann mit der eigentlichen Installation begonnen werden. Im folgenden sind die Schritte aufgelistet.

```
1. D:\
2. cd Openssl
3. set path=C:\Programme\DevStudio\VC98\Bin;%path%
4. vcvars32
5. perl Configure VC-WIN32
6. ms\do_ms
7. nmake -f ms\nt.mak
```

Für die diversen Optionen der Installation sei auf die Datei `Install` im Verzeichnis `d:\openssl` verwiesen.

Die Include-Dateien sind in `D:\Openssl\inc32\openssl`, die Libraries und die exe-Dateien in `D:\openssl\out32` zu finden.

7.4Installation unter Linux

Die Installation von OpenSSL unter Linux gestaltet sich sehr einfach. Im folgenden sind die Schritte aufgelistet.

```
1. cp  [CDROM]/OpenSSL/openssl-0.9.5a.tar.gz  /usr/local
2. cd  /usr/local
3. tar  -xzf openssl-0.9.5a.tar.gz
4. cd  openssl-0.9.5a
5. ./config
6. make
7. make  test
8. make  install
```

OpenSSL wird aus Kompatibilitätsgründen standardmässig ins Verzeichnis `/usr/local/ssl` installiert. Für die diversen Optionen der Installation sei auf die Datei `INSTALL` im Verzeichnis `/usr/local/openssl-0.9.5a` verwiesen.

Die Include-Dateien sind in `/usr/local/ssl/include`, die Libraries in `/usr/local/ssl/lib` zu finden.

Beim Kompilieren muss *gcc* mit der Option `-L/usr/local/ssl/lib -lssl –lcrypto` angegeben werden, sonst findet der Compiler/Linker die OpenSSL-Libraries nicht.

7.5Demoprogramm SSL-Handshake

Die mitgelieferten Demonstrationsprogramme bieten einen sehr oberflächlichen Einblick in die OpenSSL-Library. Nach ein paar Erweiterungen bieten sie eine zwar sehr eingeschränkte, aber funktionierende SSL-Verbindung an. Das Serverprogramm horcht auf dem Port 1111 und wartet, bis sich ein SSL-Client meldet. Dieser Client kann ein Internet-Browser sein oder das Clientprogramm. Bei Zustandekommen der SSL-Verbindung schickt der Server dem Client eine kurzen String bzw. eine HTML-Seite.

Die beiden Programme laufen unter Linux mit der oben beschriebenen Installation von OpenSSL. Der Server erwartet die Dateien *ServerCert.pem* und *ServerKey.pem*, der Client die Dateien *ClientCert.pem* und *ClientKey.pem* in demselben Verzeichnis.

Es existieren zwei Versionen des Servers:

- ***servc*** ist der Server für das Clientprogramm
- ***servb*** ist der Server für den Browser.

In den folgenden Unterkapiteln sind diese genauer erklärt. Der Source Code zu den Programmen ist in Anhang D: abgedruckt und auf der CD (siehe Anhang B:) zu finden.

7.5.1 Server

Der Server läuft auf dem Rechner *ksy107.zhwin.ch,* horcht auf dem Port 1111 und wartet, bis sich ein Client meldet und eine TCP-Verbindung aufbaut. Falls der Client kein SSL v3 unterstützt, wird die Verbindung abgebrochen. Unterstützt dieser jedoch SSL v3, wird der SSL-Handshake abgewickelt und zur Demonstration der funktionierenden Verbindung ein kurzer Datenaustausch abgewickelt. Der Server gibt ferner die aktuellen sowie eine Auflistung aller möglichen Verbindungsdaten aus (Verschlüsselungs- und Hash-Algorithmen).

Der Server muss ein Server-Zertifikat haben sowie den privaten Schlüssel dazu. Diese müssen im aktuellen Verzeichnis gespeichert sein (*ServerCert.pem, ServerKey.pem*).

Ist der Client das Client-Demoprogramm, erhält der Server einen String „Hello World!“ und schickt dem Client „I heard you“. Dann werden die SSL- und die TCP-Verbindung abgebrochen.

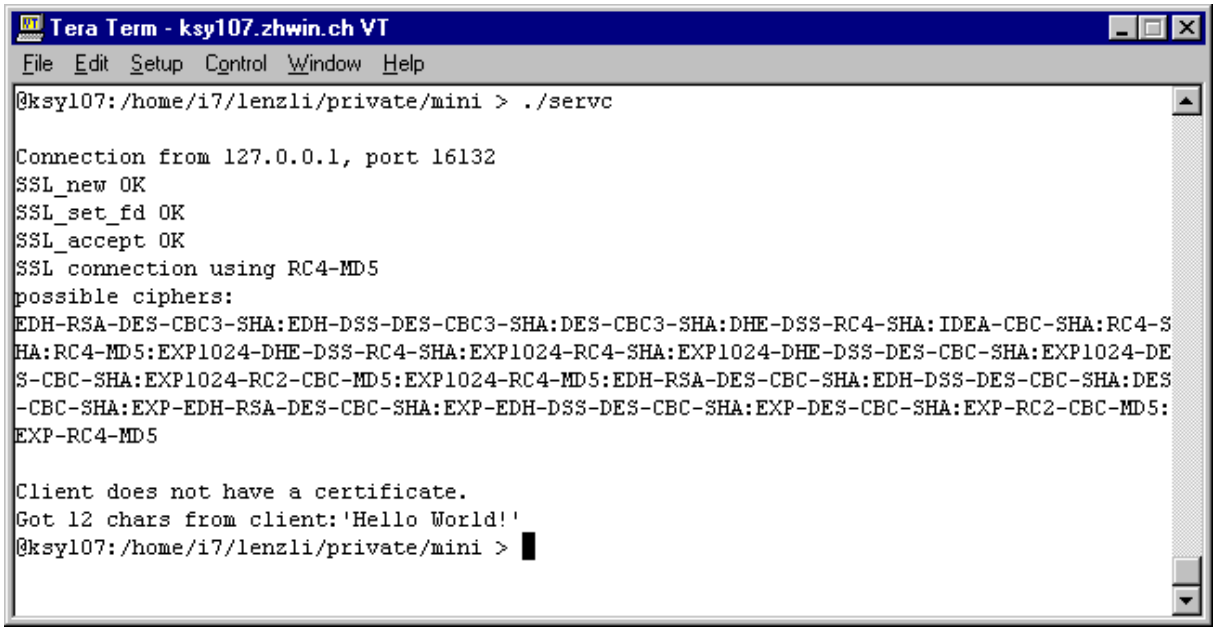

Bild 7.3: Server hat eine sichere Verbindung (RC4, MD5) mit dem Clientprogramm.

Ist der Client hingegen ein Internet-Browser (siehe Kapitel 7.5.3), erhält der Server einen HTTP-Request und schickt dem Client eine HTML-Seite zurück. Danach werden die SSL- und die TCP-Verbindung abgebrochen.

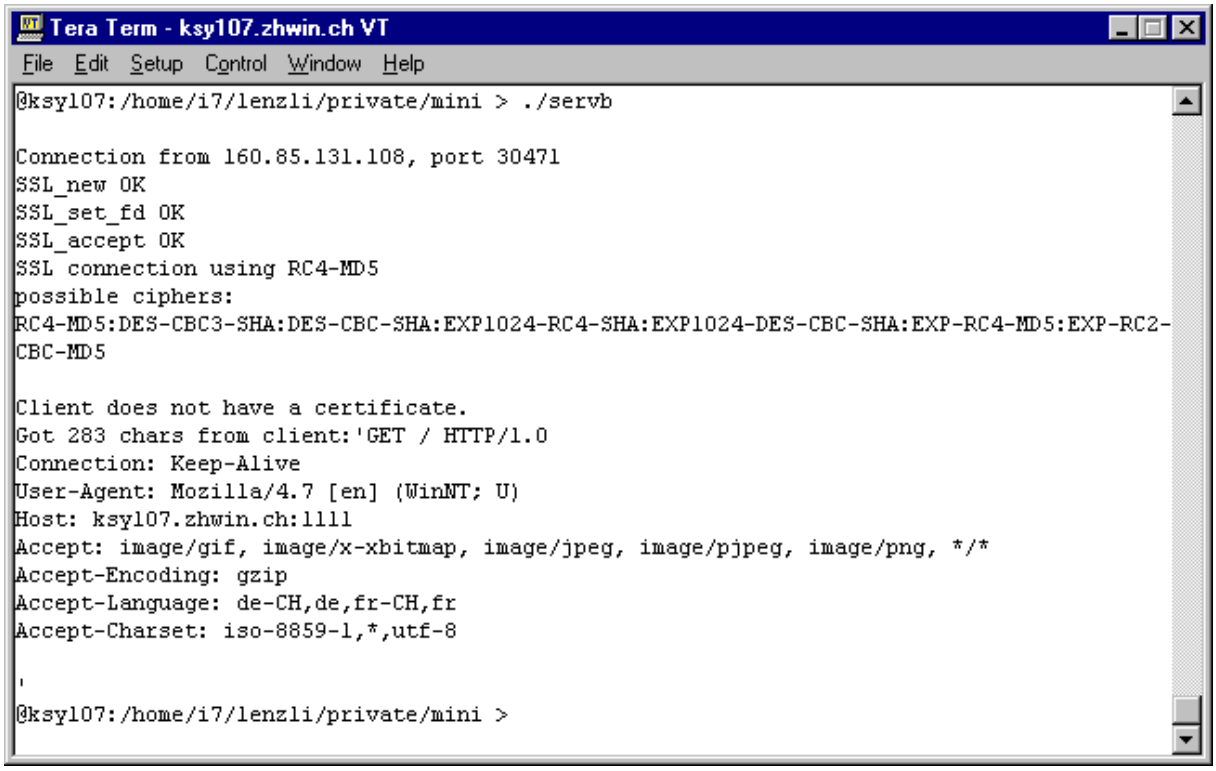

Bild 7.4: Server hat eine sichere Verbindung (RC4, MD5) mit einem Netscape-Browser.

7.5.2 Clientprogramm

Das Clientprogramm muss auf demselben Rechner laufen wie der Server. Der Client baut eine TCP-Verbindung zum Server (localhost, 127.0.0.1) auf und veranlasst den SSL-Handshake. Bei erfolgreichem Zustandekommen der SSL-Verbindung schickt der Client „Hello World!“ zum Server und gibt die Antwort des Servers aus, in diesem Fall „I heard you“. Auch das erhaltene Server-Zertifikat wird auf dem Bildschirm ausgegeben.

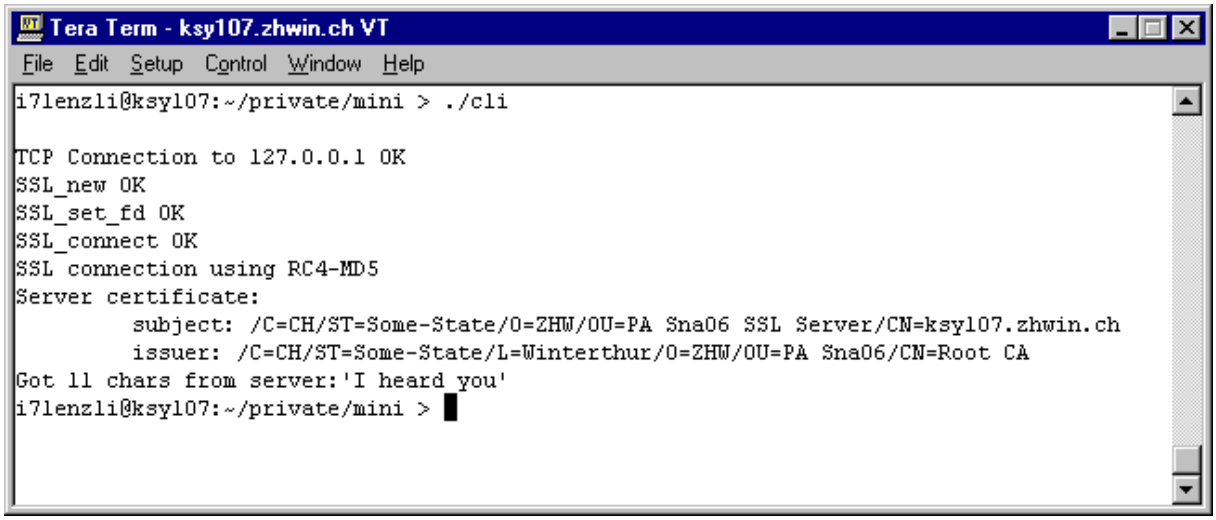

Bild 7.5: Client hat eine sichere Verbindung (RC4, MD5) zum Server.

7.5.3 Browser

Auf den Server kann auch mit einem Internet-Browser zugegriffen werden, der SSL v3 unterstützt.

Der Microsoft Internet Explorer kann nur zum Standard SSL-Port 431 eine Verbindung aufbauen, der Netscape jedoch zu einem beliebigen Port (siehe Bild 7.6). Da der Server auf Port 1111 horcht, kann somit nur Netscape verwendet werden.

Zu Beachten ist das Schlösschen im Browser, das eine verschlüsselte Verbindung signalisiert.

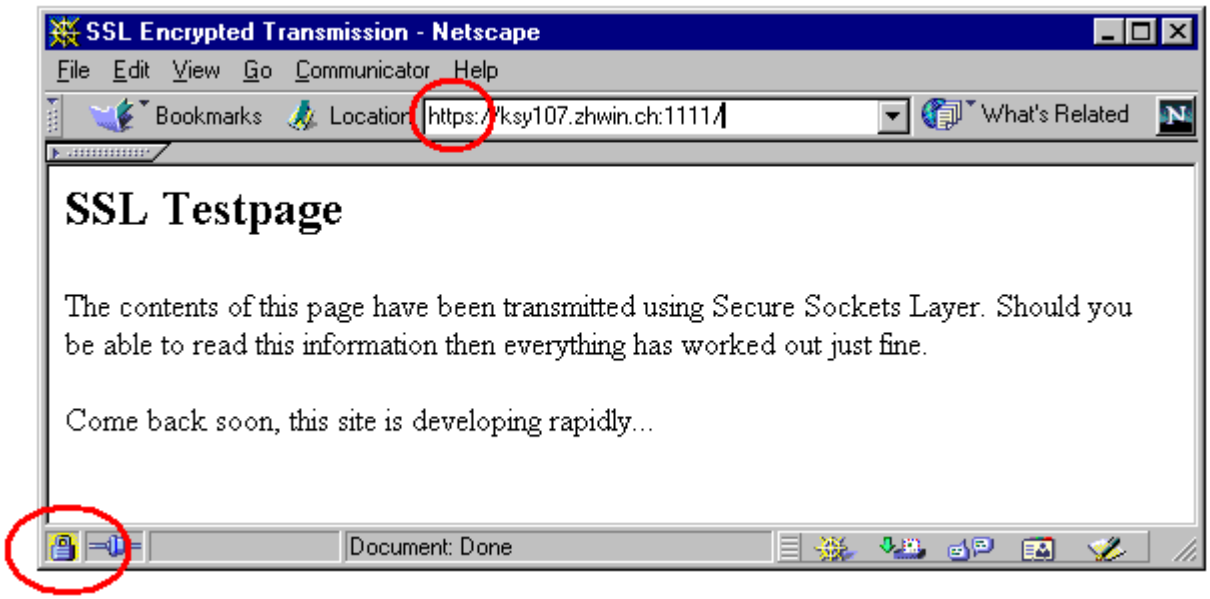

Bild 7.6: Sichere Verbindung (SSL v3) vom Netscape-Browser zum Server-Programm, das auf dem Port 1111 horcht.

8 Implementation

8.1TCP-IP API

Das integrierte *TCP-IP Application Programming Interface* des *IPC@CHIP*s ermöglicht den Zugriff auf den TCP-IP Stack für die Programmierung von eigenen Applikationen. Die einzelnen Funktionen können über den Software-Interrupt 0xAC und der jeweiligen Servicenummer im AH-Register aufgerufen werden. Diese Art von Funktionsaufrufen in Programmen ist jedoch sehr unübersichtlich und fehleranfällig:

- Nebst der korrekten Servicenummer müssen meistens zusätzlich auch noch diverse funktionsbezogene Argumente in die verschiedenen Register des Prozessors geschrieben werden → Fehleranfälligkeit.
- Der Compiler und/oder die Entwicklungsumgebung unterstützen den Programmentwickler bei der Suche nach Codierungsfehlern nicht oder nur mangelhaft.
- Der Sourcecode wird unübersichtlich und ist nicht mehr leicht zu lesen.

Als Resultat dieser Überlegungen wurde die ganze Interruptgeschichte in ein Modul (*tcpipapi.h* und *tcpip.c*) verpackt und mit einem neuen Interface ausgestattet (*tcpip.h*) und getestet. Die einzelnen Funktionen können nun ganz einfach mit Hilfe eines definierten Funktionsnamen aufgerufen werden. Als Referenz für die Definition der einzelnen Funktionen wurde die *MSDN-Library* verwendet.

Die wichtigsten Funktionen kurz beschrieben:

- **socket** – Öffnet einen neuen Socket.
- **closesocket** – Schliesst einen geöffneten Socket.
- **bind** – Verbindet einen zuvor erstellten Socket mit einer gegebenen Socket-Adresse. Die Socket-Adresse enthält die zu überwachende Portnummer und die IP-Adresse des Servers.
- **listen** – Programm beginnt mit der Überwachung des im vorausgegangenen bind-Aufruf genannten Ports.
- **accept** – Gibt die Kontrolle zurück, wenn ein Client eine Verbindung zu diesem Socket herstellt, und sendet dann einen neuen Socket, der der neuen Verbindung entspricht.
- **connect** – Gibt die Kontrolle zurück, sobald die Verbindung zum Server hergestellt ist.
- **send** – Senden von Daten.
- **recv** – Empfangen von Daten.

In Anhang D: findet sich eine komplette Beschreibung der einzelnen Funktionen sowie die dazugehörenden Listings.

8.2OpenSSL für IPC@CHIP DOS

Das derzeit verwendete BIOS v0.66 des IPC@CHIP benötigt knapp die Hälfte des zur Verfügung stehenden FlashROM. Das heisst, benutzerspezifische Applikationen dürfen maximal 256 KByte gross sein!

Das OpenSSL-Build mit all seinen Features benötigt jedoch weit mehr als ein MByte an Speicher. Um ein Minimum an Programmgrösse zu erreichen, wurde deshalb auf die meisten Digest – und Symmetric Cipher – Implementationen verzichtet, beim SSL haben wir uns auf die ausschliessliche Unterstützung der Version 3 geeinigt.

In Tabelle 8.1 sind die Verwendeten OpenSSL-Komponenten aufgelistet.

Authentication Codes, Hash Functions	md5, sha
Symmetric Ciphers	rc4
Public Key Cryptography and Key Agreement	rsa
Certificates	x509, x509v3
Data Encoding	ans1, pkcs7, pkcs12
Unterstützende SSL Version	ssl3
Diverse Interne Funktionen	...

Tabelle 8.1: Verwendete OpenSSL-Komponenten

8.2.1 Makefile

Achtung: Das in der OpenSSL-Distribution enthaltene Perl-Skript *Configure* zum Erstellen von Makefiles ist leider nicht in der Lage, funktionierende Makefiles für die verwendete Version von Borland C++ 3.0 (BC) zu erstellen.

Aus diesem Grund wurde zuerst mit Hilfe von Active-Perl ein entsprechendes Makefile für die Windows NT-Plattform erstellt, das dann für BC angepasst werden musste. Da unter DOS die Dateinamenlänge auf acht Zeichen beschränkt ist, mussten einige *.h und *.c Dateien entsprechend geändert werden. Die von Perl generierten Makefiles erstellen standardmässig die beiden Programmbibliotheken ssl.lib und crypto.lib. Letztere wurde wegen ihrer Codegrösse (über ein MByte gross) in mehrere, nach Funktionen getrennte, kleinere Programmbibliotheken zerlegt.

Das an Borland C++ 3.0 angepasste Makefile *opsslbc.mak* kompiliert alle nötigen Sourcefiles und erstellt anschliessend die einzelnen Programmbibliotheken. Diese können auf einfache Weise in einem konkreten Softwareprojekt eingebunden werden. Eine vollständige Liste der Optionen ist in [2] ab Seite 619 oder im BC Helpfile zu finden.

Das komplette Listing des Makefiles *opsslbc.mak* ist auf der CD vorhanden (siehe Anhang B:).

8.2.1.1 Anwenden von *opsslbc.mak*

Das Makefile *opsslbc.mak* enthält diverse Pfadangaben, welche auf BC und OpenSSL verweisen. Damit die .mak-Datei fehlerfrei ausgeführt werden kann, müssen BC und OpenSSL folgendermassen installiert worden sein:

- BC im Verzeichnis `C:\BorlandC`
- OpenSSL im Verzeichnis `D:\OpenSSL`
- *opsslbc.mak* befindet sich im Verzeichnis `D:\OpenSSL\ms`.

Nun kann in einem Konsolen-Fenster mit dem Dienstprogramm *make* das Makefile *opsslbc.mak* kompiliert werden. Im folgenden sind die Schritte aufgelistet.

```
1. d:\
2. set path=c:\borlandc\bin;d:\openssl;d:\openssl\ms;%path%
3. cd openssl\ms
4. make -f opsslbc.mak
```

Die Include-Dateien sind in `d:\openssl\inc16bc\openssl`, die Libraries in `d:\openssl\out16bc` zu finden.

8.2.1.2 Anwenden der erzeugten Programmbibliotheken

Mit dem Linker können die einzelnen Programmbibliotheken in ein Programm eingebunden werden. Im folgenden Beispiel wird *tlink* mit einer Reihe von Programmargumenten aufgerufen. Das Argument *libs.txt* bezeichnet eine externe Datei, in der alle zu verwendenden *.lib aufgelistet sind. In Bild 8.1 ist der Inhalt von libs.txt dargestellt.

Die genauen Syntax für *tlink* ist erhältlich in der BC-Help oder durch den Aufruf von *tlink* (ohne Argumente).

Beispiel mit *tlink*:

```
tlink /C/c/m/s/s/Lc:\borlandc\lib;d:\openssl\out16bc serv.obj
tcpip.obj,,, @libs.txt
```

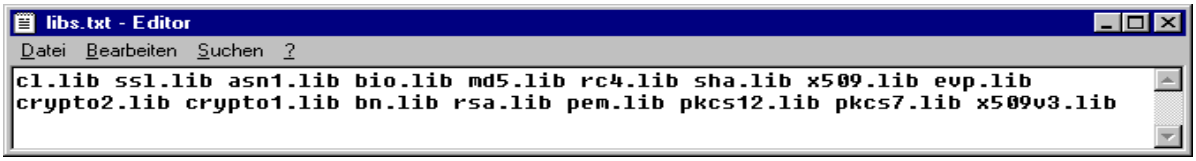

Bild 8.1: Datei libs.txt mit Programmbibliotheken.

8.2.2 Borland C++ 3.0

Das grösste Problem der verwendeten Dienstprogramme aus der BC-Distribution ist, dass ihnen beim Programmstart nur eine begrenzte Anzahl von Programmargumenten übergeben werden kann. Dieses Problem kann umgangen werden, in dem man die meisten Argumente in eine Textdatei verpackt und als solche dem Programm übergibt.

Eine vollständige Liste der Argumente der folgenden Werkzeuge sind beschrieben in [2] ab Seite 619.

- **Dienstprogramm *make***
  Mit dem Dienstprogramm *make* kann auf bequeme Weise die bei der Programmentwicklung notwendigen (u.U. recht häufig wiederkehrenden) Zwischenschritte (Compilierung, Library-Erstellung, Linken) automatisch veranlasst werden. Hierzu werden die gegenseitigen Abhängigkeiten zwischen den verschiedenen Quell-, Header-, Object- und ausführbaren Dateien in einem sog. *Makefile* festgelegt. Anhand dieses Files und des jeweiligen Datums der letzten Änderung der entsprechenden Quelldatei, Headerdatei usw. werden die zum Neuaufbau des Programms notwendigen Prozesse automatisch durch *make* gestartet, andererseits werden (bei korrektem Aufbau des Makefiles) keine unnötigen Compilierungs- oder Linkvorgänge unternommen.

  Die genauen Syntax für *make* ist erhältlich in der BC-Help oder durch den Aufruf von *make -?*.

- **Compiler *bcc***
  Compiler, für die Übersetzung von Sourcecode (*.c, *.cpp) in Binärcode (*.obj).

  Die genauen Syntax für *bcc* ist erhältlich in der BC-Help oder durch den Aufruf von *bcc* (ohne Argumente).

- **Library Builder *tlib***
  Hilfsprogramm zur Erstellung und Verwaltung von Programmbibliotheken (*.lib).

  Die genauen Syntax für *tlib* ist erhältlich in der BC-Help oder durch den Aufruf von *tlib* (ohne Argumente).

- **Linker *tlink***
  Hilfsprogramm für das Linken und Erstellen von ausführbarem Programmcode (*.com, *.exe).

  Die genauen Syntax für *tlink* ist erhältlich in der BC-Help oder durch den Aufruf von *tlink* (ohne Argumente).

9 Ausblick

Im Rahmen dieser Projektarbeit sollte abgeklärt werden, ob die als C-Source Code verfügbare OpenSSL-Library auf den IPC@CHIP portiert werden kann. Insbesondere sollte die Anbindung an das TCPIP-API des IPCs untersucht werden.

Die Anbindung von OpenSSL an das TCPIP-API des IPC@CHIP ist kein Problem. Wir haben die nötigen Anpassungen in einem Interface implementiert (siehe Kapitel 8.1 und Anhang D:).

Die Portierung von OpenSSL auf die IPC@CHIP-Plattform ist nach unserem Ermessen grundsätzlich möglich. Folgenden Punkten ist jedoch Beachtung zu schenken:

1. Mit Hilfe unseres Makefiles (`[CDROM]\OpenSSL für IPC_CHIP\opsslbc.mak`) konnte die abgespeckte Version von OpenSSL (siehe Kapitel 8.2) kompiliert und gelinkt werden. Ein erstelltes Testprogramm wurde aber zu gross, und konnte nicht auf den IPC geladen werden. Es gilt also Wege zu finden, die Funktionalität und somit auch die Grösse der ausführbaren Datei zu reduzieren.
2. Der Architekturunterschied gibt auch Probleme auf. Der Datentyp int ist in einer Linuxumgebung, wo OpenSSL herkommt, 4 Byte lang, unter DOS jedoch nur 2 Byte. Diese Problematik ist insofern nicht sehr gross, da unter DOS ein long verwendet werden kann. Das hat aber zur Folge, dass die gesamte OpenSSL-Library angepasst werden muss, was unter Umständen in Arbeit ausarten kann.
3. Die Performance konnten wir nicht testen, da die Portierung noch nicht vollzogen ist. Sie dürfte aber bei einem 80186 mit 20 MHz Taktfrequenz am unteren Limit ausfallen. Hier muss getestet werden, ob eine benutzerfreundliche Geschwindigkeit des Datenflusses erreicht werden kann.

Das in dieser Projektarbeit gewonne Wissen und die Grundlagen bieten eine gute Basis für eine konkrete Portierung. Vor allem die Installation der Entwicklungsumgebung (Hard- und Software) ist mit Hilfe unserer Anleitung schnell und problemlos durchführbar.

Die Chancen einer Portierung von OpenSSL und eine konkrete Anwendung müssen realistisch gesehen werden. Es gibt viele mögliche Szenarien. Zum Beispiel könnte die Portierung erfolgreich verlaufen, aber die Performance genügt nicht. Oder die Anwendung könnte zu gross sein für die 256 Kbyte Flash-Speicher. Dann müsste allenfalls ein eigenes Challenge-Protokoll entworfen werden, das aber bestimmt etliche Funktionen der OpenSSL-Library nutzen kann.

10 Schlusswort

Das Thema Kryptographie und Sicherheit im Internet ist ein sehr interessantes Thema und hat uns beide sehr fasziniert. Im Rahmen dieser Projektarbeit konnten wir endlich unser Backgroundwissen aus dem Vertiefungsfach Kommunikationssysteme sinnvoll einsetzen und auch zielgerichtet erweitern.

Ebenfalls sehr spannend war die Arbeit mit einer Opensource-Distribution wie OpenSSL. An diesem komplexen Softwaregebilde gibt es eigentlich nur eines zu beklagen: Die Dokumentation ist zum grössten Teil sehr düftig ausgefallen.

Der IPC@CHIP verdeutlichte uns, dass für moderne Anwendungen nicht zwangsweise auch die neuesten Technologien eingesetzt werden müssen. So ist es mit diesem Chip möglich, über eine integrierte Ethernetschnittstelle eine TCP/IP Verbindung aufzubauen und sich in das Intranet/Internet einzuklinken.

Unseren gewonnen Erkentnissen zu Folge ist das hauptsächliche Anwendungsgebiet des IPC@CHIP heutzutags im firmeninternen Intranet zu finden, wo von einem beliebigen PC aus per Browser auf die Daten einer kleinen Steuerung zugegriffen werden kann. Sobald das weltweite Internet aber ausreichende IP-Adressen zur Verfügung stellt (IPv6), ist es vorstellbar, dass mit dieser Variante auch der Internet-Zugriff auf digitale IOs realisierbar wird, etwa auf Licht, Küchengeräte oder Heizung im Haus. Dann kommt auch die Sicherheitsproblematik voll zum Tragen.

Positiv überrascht waren wir von der von uns verwendeten, schon etwas in die Jahre gekommenen Entwicklungsumgebung. Irgendwo zuhinterst in einer Schublade fanden wir ein paar verstaubte Disketten, auf denen Borland C++ 3.0 für DOS darauf wartete, wieder zum Leben erweckt zu werden. Wir gaben ihm eine Chance, und siehe da: das Alter einer Software muss nicht umgekehrt proportional sein zu den Diensten, die sie leistet. BC hatte zwar so seine Tücken, vor allem die kurzen Dateinamen gaben Probleme auf, aber dafür war die integrierte Hilfe zu den enthaltenen Softwarekomponenten ausgezeichnet.

Die Teamarbeit gestaltete sich als sehr interessant und produktiv.

Wir danken Herrn Steffen für die hilfreiche Unterstützung, die wir in diesem Praktikum hatten.

Anhang

A: Konfiguration des IPC@CHIP

Der IPC@CHIP ist als DHCP Client konfiguriert und ist unter *mini.zhwin.ch* oder direkt mit der IP-Adresse *160.85.134.67* ansprechbar.

SC12 Seriennummer	0016F
MAC Adresse	00 30 56 F0 01 6F
IP Adresse	160.85.134.67
Subnetzmaske	255.255.240.0
Gateway	160.85.128.1

Tabelle **A:**.1: Konfiguration des IPC@CHIP

Das INI-File *chip.ini* bestimmt die verschiedenen Modi des IPC@CHIP. Mit folgendem *chip.ini* wurde gearbeitet:

```
[IP]
DHCP=1

[DEVICE]
NAME=MINI

[WEB]
ENABLE=1
ROOTDIR=WEB

[FTP]
USER0=BEN
PASSWORD0=BL1203
USER1=ANDY
PASSWORD1=ZIA

[TELNET]
USER0=BEN
PASSWORD0=BL1203
USER1=ANDY
PASSWORD1=ZIA
```

B: Inhaltsverzeichnis der CD

Bild B:1 zeigt den Inhalt der CD-ROM.

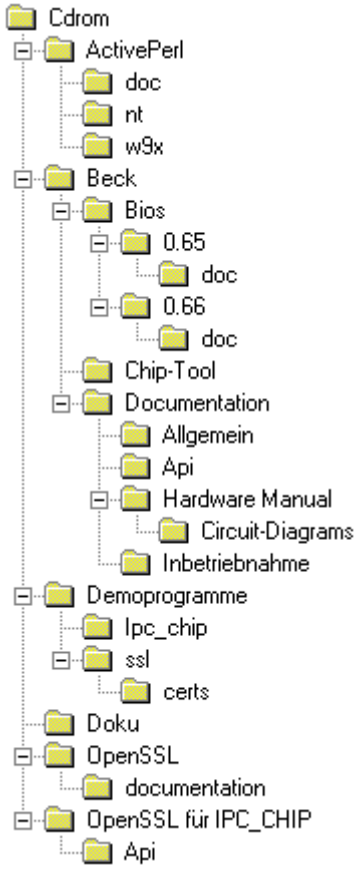

Bild B:1: Inhaltsverzeichnis der CDROM

C: Literaturverzeichnis

[1] Thomas, Stephen.
***SSL and TLS Essentials**. Securing the Web.*
Wiley Computer Publishing, ISBN 0-471-38354-6

[2] Achtert, Werner.
***Das grosse Buch zu C++**. Erfolgreich programmieren mit C++.*
Data Becker, ISBN 3-8158-1177-5

[3] http://www.beck-ipc.com

[4] http://www.openssl.org

[5] http://www.pca.dfn.de/dfnpca/certify/ssl/handbuch

[6] How SSL Works
http://developer.netscape.com/tech/security/ssl/howitworks.html

D: Source Code

D.1 serv.c

```
#include <stdlib.h> // bl
#include <unistd.h> // bl

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // bl
#include <memory.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <openssl/rsa.h>       /* SSLeay stuff */
#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>
#include <openssl/err.h>


#define HTML_PAGE "<HTML><HEAD><TITLE>SSL Encrypted
Transmission</TITLE></HEAD><BODY><P><BIG><BIG><B>SSL Testpage</B></BIG></BIG><P><P>The
contents of this page have been transmitted using Secure Sockets Layer. Should you be able to
read this information then everything has worked out just fine.<BR><BR>Come back soon, this
site is developing rapidly...</BODY></HTML>"

////////////////////////////////////////////////////
#ifdef CLIENT_IS_BROWSER
#define ANSWER_STRING HTML_PAGE
#else
#define ANSWER_STRING "I heard you"
#endif
////////////////////////////////////////////////////



#define PORT_NUMBER 1111

// define HOME to be dir for key and certificate files
#define HOME "./"
#define CERT_PATH  HOME "ServerCert.pem"
#define KEY_PATH  HOME  "ServerKey.pem"

#define CHK_NULL(x) if ((x)==NULL) exit (1)
#define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }
#define CHK_SSL(err) if ((err)==-1) { ERR_print_errors_fp(stderr); exit(2); }

int main ()
{
  char tempstr[500];
  int pid;
  int err;
```

```
int listen_sd;
int sd;
struct sockaddr_in sa_serv;
struct sockaddr_in sa_cli;
size_t client_len;
SSL_CTX* ctx;
SSL*     ssl;
X509*    client_cert;
char*    str;
char     buf [4096];
SSL_METHOD *meth;

char clientaddr[15] = "0.0.0.0";


/* SSL preliminaries. We keep the certificate and key with the context. */

SSL_load_error_strings();
SSLeay_add_ssl_algorithms();
meth = SSLv3_server_method();
ctx = SSL_CTX_new (meth);
if (!ctx) {
  ERR_print_errors_fp(stderr);
  exit(2);
}

if (SSL_CTX_use_certificate_file(ctx, CERT_PATH, SSL_FILETYPE_PEM) <= 0) {
  ERR_print_errors_fp(stderr);
  exit(3);
}

if (SSL_CTX_use_PrivateKey_file(ctx, KEY_PATH, SSL_FILETYPE_PEM) <= 0) {
  ERR_print_errors_fp(stderr);
  exit(4);
}

if (!SSL_CTX_check_private_key(ctx)) {
  fprintf(stderr,"Private key does not match the certificate public key\n");
  exit(5);
}

/* ------------------------------------------------ */
/* Prepare TCP socket for receiving connections */

listen_sd = socket (AF_INET, SOCK_STREAM, 0);
CHK_ERR(listen_sd, "socket");

memset (&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family      = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port        = htons ( PORT_NUMBER );          /* Server Port number */

err = bind(listen_sd, (struct sockaddr*) &sa_serv,sizeof (sa_serv));
CHK_ERR(err, "bind");

/* Receive a TCP connection. */
```

```
//  while (1) {
      err = listen (listen_sd, 5);
      CHK_ERR(err, "listen");
//      pid = fork();
//      if (pid == 0) break;
//  }

  client_len = sizeof(sa_cli);
  sd = accept (listen_sd, (struct sockaddr*) &sa_cli, &client_len);
  CHK_ERR(sd, "accept");
  close (listen_sd);

  strcpy( clientaddr ,inet_ntoa(sa_cli.sin_addr));
  printf ("\nConnection from %s, port %i\n", clientaddr, sa_cli.sin_port);

  /* ---------------------------------------------- */
  /* TCP connection is ready. Do server side SSL. */

  ssl = SSL_new (ctx);
  SSL_set_cipher_list( ssl, SSL_TXT_RC4_128_WITH_MD5 );
  CHK_NULL(ssl);
  printf( "SSL_new OK\n" ); // bl
  SSL_set_fd (ssl, sd);
  printf( "SSL_set_fd OK\n" ); // bl
  err = SSL_accept (ssl);
  CHK_SSL(err);
  printf( "SSL_accept OK\n" ); // bl

  /* Get the cipher - opt */

  printf ("SSL connection using %s\n", SSL_get_cipher (ssl));

  printf("possible ciphers:\n%s\n\n", SSL_get_shared_ciphers(ssl,tempstr,500));

  /* Get client's certificate (note: beware of dynamic allocation) - opt */

  client_cert = SSL_get_peer_certificate (ssl);
  if (client_cert != NULL) {
    printf ("Client certificate:\n");

    str = X509_NAME_oneline (X509_get_subject_name (client_cert), 0, 0);
    CHK_NULL(str);
    printf ("\t subject: %s\n", str);
    Free (str);

    str = X509_NAME_oneline (X509_get_issuer_name  (client_cert), 0, 0);
    CHK_NULL(str);
    printf ("\t issuer: %s\n", str);
    Free (str);

    /* We could do all sorts of certificate verification stuff here before
       deallocating the certificate. */

    X509_free (client_cert);
  } else
    printf ("Client does not have a certificate.\n");
```

```
  /* DATA EXCHANGE - Receive message and send reply. */

  err = SSL_read (ssl, buf, sizeof(buf) - 1);
  CHK_SSL(err);
  buf[err] = '\0';
  printf ("Got %d chars from client:'%s'\n", err, buf);

  err = SSL_write (ssl, ANSWER_STRING, strlen( ANSWER_STRING ));
  CHK_SSL(err);

  /* Clean up. */

  close (sd);
  SSL_free (ssl);
  SSL_CTX_free (ctx);

  return( 0 );
}
```

D.2 cli.c

```
#include <stdlib.h> // bl
#include <unistd.h> // bl

#include <stdio.h>
#include <memory.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define SERVER_IP "127.0.0.1"  // localhost

// define HOME to be dir for key and certificate files
#define HOME "./"
#define CERT_PATH  HOME "ClientCert.pem"
#define KEY_PATH  HOME  "ClientKey.pem"


#define CHK_NULL(x) if ((x)==NULL) exit (1)
#define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }
#define CHK_SSL(err) if ((err)==-1) { ERR_print_errors_fp(stderr); exit(2); }

int main ()
{
  int err;
  int sd;
```

```
struct sockaddr_in sa;
SSL_CTX* ctx;
SSL*     ssl;
X509*    server_cert;
char*    str;
char     buf [4096];
SSL_METHOD *meth;

SSLeay_add_ssl_algorithms();
meth = SSLv3_client_method();
SSL_load_error_strings();
ctx = SSL_CTX_new (meth);

CHK_NULL(ctx);

CHK_SSL(err);

/* ---------------------------------------------- */
/* Create a socket and connect to server using normal socket calls. */

sd = socket (AF_INET, SOCK_STREAM, 0);
CHK_ERR(sd, "socket");

memset (&sa, '\0', sizeof(sa));
sa.sin_family      = AF_INET;
sa.sin_addr.s_addr = inet_addr (SERVER_IP);   /* Server IP */
sa.sin_port        = htons     (1111);        /* Server Port number */

err = connect(sd, (struct sockaddr*) &sa, sizeof(sa));
CHK_ERR(err, "connect");

printf( "\nTCP Connection to %s OK\n", SERVER_IP ); // bl

/* ---------------------------------------------- */
/* Now we have TCP conncetion. Start SSL negotiation. */

ssl = SSL_new (ctx);
CHK_NULL(ssl);
printf( "SSL_new OK\n" ); // bl
SSL_set_fd (ssl, sd);
printf( "SSL_set_fd OK\n" ); // bl
err = SSL_connect (ssl);
CHK_SSL(err);
printf( "SSL_connect OK\n" ); // bl

/* Following two steps are optional and not required for
   data exchange to be successful. */

/* Get the cipher - opt */

printf ("SSL connection using %s\n", SSL_get_cipher (ssl));

/* Get server's certificate (note: beware of dynamic allocation) - opt */

server_cert = SSL_get_peer_certificate (ssl);
CHK_NULL(server_cert);
printf ("Server certificate:\n");
```

```
  str = X509_NAME_oneline (X509_get_subject_name (server_cert),0,0);
  CHK_NULL(str);
  printf ("\t subject: %s\n", str);
  Free (str);

  str = X509_NAME_oneline (X509_get_issuer_name  (server_cert),0,0);
  CHK_NULL(str);
  printf ("\t issuer: %s\n", str);
  Free (str);

  /* We could do all sorts of certificate verification stuff here before
     deallocating the certificate. */

  X509_free (server_cert);

  /* --------------------------------------------------- */
  /* DATA EXCHANGE - Send a message and receive a reply. */

  err = SSL_write (ssl, "Hello World!", strlen("Hello World!"));
  CHK_SSL(err);

  err = SSL_read (ssl, buf, sizeof(buf) - 1);
  CHK_SSL(err);
  buf[err] = '\0';
  printf ("Got %d chars from server:'%s'\n", err, buf);
  SSL_shutdown (ssl);  /* send SSL/TLS close_notify */

  /* Clean up. */

  close (sd);
  SSL_free (ssl);
  SSL_CTX_free (ctx);

  return( 0 );
}
```

D.3 tcpipapi.h

```
/*-----------------------------------------------------------------------------------------------
            D E F I N I T I O N   M O D U L E
-----------------------------------------------------------------------------------------------
PRODUCT:    Project SNA06: Mini Web Server with SSL, feasibility study

TITLE:      -
-----------------------------------------------------------------------------------------------
            Version:                        Author:                 Date:
IDENT:      tcpipapi.h.10                   A.Zingg                 17. July 2000
              (created)                     B.Lenzlinger
-----------------------------------------------------------------------------------------------
PURPOSE:    Interface definition for TCPIP Socket-Interface.
            The TCPIP-API provides the interrupt 0xAC with a service number in the high byte
            of the AX register (AH). The interface enables the access to the TCP/IP-stack of
            the IPC@Chip for programming own TCP/IP-Applications.
```

```
            Use memory model large!!

REFERENCES: IPC@CHIP Documentation. BIOS Ver. 0.66

COMPILER:   bcc (Borland C++ Ver. 3.1)
-------------------------------------------------------------------------------------------*/

//  ------------------------------------------------
//  1.    I N T E R F A C E    D E F I N I T I O N S
//  ------------------------------------------------
#ifndef _TCPIP_API_H__
#define _TCPIP_API_H__

//-------------------------------------------------------------------------------------------


//  ----------------------------------------
//  2.    G L O B A L    D E F I N I T I O N S
//  ----------------------------------------

//  2.1 BSD Socket Definitions
//  --------------------------
#define AF_INET  2
#define PF_INET  AF_INET

#define SOCK_STREAM     1     // Stream socket, TCP.
#define SOCK_DGRAM      2     // Datagram socket, UDP.

#define MSG_BLOCKING          0x0000  // This message should be blocking.
#define MSG_TIMEOUT           0x0001  // Wake up from recv after we have timed out.
#define MSG_DONTWAIT          0x0080  // This message should be nonblocking.

//  2.1 BSD Structure Definitions
//  -----------------------------
struct sockaddr
{
  unsigned char sa_len;      // Total Length.
  unsigned char sa_family;   // Address Family AF_xxx.
  char sa_data[14];          // Up to 14 bytes of protocol specific address.
};

struct in_addr
{
  unsigned long s_addr;      // 32bit netis/hostid address in network byte order.
};

struct sockaddr_in
{
  short           sin_family; // Address family (must be AF_INET).
  unsigned int    sin_port;   // 16bit Port Number in network byte order (IP port).
  struct  in_addr sin_addr;   // 32bit netid/hostid in network byte order (IP address).
  char            sin_zero[8];// Unused.
};

//  2.3 API Interface-Functionnumbers
//  ---------------------------------
#define API_OPENSOCKET       1   // Open a Socket.
```

```
#define API_CLOSESOCKET      2   // Close a Socket.
#define API_BIND             3   // Bind a unnamed socket with an address and portnumber.
#define API_CONNECT          4   // TCP client only! Connect to another socket.
#define API_RECVFROM         5   // UDP only! Receive message from another socket.
#define API_SENDTO           6   // UDP only! Transmit message to another transport end-point.
#define API_HTONS            7   // Convert byte order.
#define API_INETADDR         8   // Converts an dotted decimal IP-String to an unsigned long.
#define API_SLEEP            9   // The application sleeps for given milliseconds
#define API_MALLOC           10  // Alloc a buffer.
#define API_FREE             11  // Free an allocated buffer.
#define API_GETRCV_BYTES     12  // Get the number of bytes on a socket, waiting for read.
#define API_ACCEPT           13  // TCP server only! Accept the next incoming connection.
#define API_LISTEN           14  // TCP server only! Listening for incoming connections.
#define API_SEND             15  // TCP only! Transmit a message.
#define API_RECV             16  // TCP only! Receive message.
#define API_INETTOASCII      17  // Convert an IP address to an IP string.
#define API_RESETCONNECTION  18  // TCP only! Abort a connection on a socket.
#define API_SETLINGER        19  // TCP only! Set linger time on close.
#define API_SETREUSE         20  // TCP server only! Set reuse option on a listening socket.

//  2.4 API Errorcodes
//  ------------------
#define API_NOT_SUPPORTED       -2
#define API_ERROR               -1
#define API_ENOERROR             0

//  2.4 API Send and Receive Parameter Structure Definition
//  ------------------------------------------------------
// This structure is needed for API-receivecalls.
//
struct recv_params
{
  char* bufferPtr;           // Buffer for recveived data.
  int bufferLength;          // Expected length.
  int flags;                 // Blocking or  dontwait.
  struct sockaddr* fromPtr;  // Only needed for UDP! Contains ip and portnumber.
  int* fromlengthPtr;        // Only needed for UDP! Length of fromPtr.
  unsigned long timeout;     // Timeout milliseconds.
};

// This structure is needed for API-sendcalls.
//
struct send_params
{
  char* bufferPtr;         // Pointer to sendbuffer.
  int bufferLength;        // Length of buffer.
  int flags;               // Sendflags --> always zero.
  struct sockaddr* toPtr;  // Only needed for UDP! Contains dest.ip and dest.portnumber.
  int* tolengthPtr;        // Only needed for UDP! Length of toPtr.
};
//-----------------------------------------------------------------------------------------

#endif // _TCPIP_API_H__
// end tcpipapi.h
```

D.4 tcpip.h

```
/*-------------------------------------------------------------------------------------------
           D E F I N I T I O N   M O D U L E
-------------------------------------------------------------------------------------------
PRODUCT:    Project SNA06: Mini Web Server with SSL, feasibility study

TITLE:      -
-------------------------------------------------------------------------------------------
            Version:                        Author:                 Date:
IDENT:      tcpip.h.10                      A.Zingg                 17. July 2000
              (created)                     B.Lenzlinger
-------------------------------------------------------------------------------------------
PURPOSE:    Prototypes constants for TCPIP API socket functions.

            Use memory model large!!

REFERENCES: - IPC@CHIP Documentation. BIOS Ver. 0.66
            - MSDN Library Visual Studio 6.0 release, prototype definition for
              socketfunctions

COMPILER:   bcc (Borland C++ Ver. 3.1)
-------------------------------------------------------------------------------------------*/

//  ------------------------------------------------
//  1.    I N T E R F A C E    D E F I N I T I O N S
//  ------------------------------------------------
#ifndef _TCPIP_H__
#define _TCPIP_H__

#include "TCPIPAPI.H"
//-------------------------------------------------------------------------------------------


//  ----------------------------------------
//  2.    G L O B A L    D E F I N I T I O N S
//  ----------------------------------------

//  2.1 Global Constants
//  --------------------
#define TCPIPVECT  0xAC     // Interrupt Vector for TCPIP-API-Calls

//  2.2 Global Prototypes
//  ---------------------
// standard socket functions
//
int socket(int af, int type, int protocol);
int closesocket(int s);
int bind(int s, const struct sockaddr* name, int namelen);
int listen(int s, int backlog);
int accept(int s, struct sockaddr* addr, int* addrlen);
int connect(int s, const struct sockaddr* name, int namelen);
int send(int s, char* buf, int len, int flags);
int recv(int s, char* buf, int len, int flags);
int sendto(int s, const char* buf, int len, int flags, const struct sockaddr* to, int tolen);
int recvfrom(int s, char* buf, int len, int flags, struct sockaddr* from, int* fromlen);
unsigned int htons(unsigned int hostshort);
unsigned long inet_addr(const char* cp);
```

```
int inet_ntoa(unsigned long* in, char* out);

// special socket functions
//
void TCPIP_Sleep(unsigned int howlong);
int TCPIP_GetWaitingBytes(int sd, int* error);
int TCPIP_ResetConnection(int sd, int* error);
int TCPIP_SetLinger(int sd, int seconds, int* error);
int TCPIP_SetReuse(int sd,int* error);
//---------------------------------------------------------------------------

#endif // _TCPIP_H__
//end tcpip.h
```

D.5 tcpip.c

```
/*---------------------------------------------------------------------------
           I M P L E M E N T A T I O N   M O D U L E
---------------------------------------------------------------------------
PRODUCT:    Project SNA06: Mini Web Server with SSL, feasibility study

TITLE:      -
---------------------------------------------------------------------------
            Version:                        Author:                 Date:
IDENT:      tcpip.c.10                      A.Zingg                 17. July 2000
              (created)                     B.Lenzlinger
---------------------------------------------------------------------------
PURPOSE:    TCPIP API socket functions.

            Use memory model large!!

REFERENCES: -

COMPILER:   bcc (Borland C++ Ver. 3.1)
---------------------------------------------------------------------------*/

//  ------------------------------------------------
//  1.    I N T E R F A C E    D E F I N I T I O N S
//  ------------------------------------------------
#include <DOS.H>
#include <STDLIB.H>
#include "TCPIPAPI.H"
#include "TCPIP.H"
//---------------------------------------------------------------------------


//  -------------------------------------
//      3.    G L O B A L   F U N C T I O N S
//      -------------------------------------
// standard socket functions
//

//---------------------------------------------------------------------------
// socket.
//
```

```
// Descriptinon:
//      The  socket function creates a socket that is bound to a specific service provider.
//
// Arguments:
//      af          - [in] An address family specification (unused).
//      type        - [in] A type specification for the new socket:
//                    SOCK_DGRAM -> UDP or SOCK_STREAM -> TCP.
//      type        - [in] A particular protocol to be used with the socket that is specific
//                    to the indicated address family (unused).
//
// Return Value:
//      if SUCCESS  - Socketdescriptor.
//      else        - API_ERROR.
//-----------------------------------------------------------------------------------------
int socket(int af, int type, int protocol)
{
  union REGS inregs;
  union REGS outregs;

  inregs.h.ah = API_OPENSOCKET;
  inregs.h.al = type;
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx == (unsigned int)API_ERROR) {

    return API_ERROR;
  }

  return outregs.x.ax;
} // socket.
//-----------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------
// closesocket.
//
// Descriptinon:
//      The closesocket function closes an existing socket.
//
// Arguments:
//      s           - [in] A descriptor identifying a socket to close.
//
// Return Value:
//      Success             - API_ENOERROR.
//      Socket call failed  - API_ERROR.
//-----------------------------------------------------------------------------------------
int closesocket(int s)
{
  union REGS inregs;
  union REGS outregs;

  inregs.h.ah = API_CLOSESOCKET;
  inregs.x.bx = s;
  int86(TCPIPVECT, &inregs, &outregs);

  return outregs.x.dx;
} // closesocket.
//-----------------------------------------------------------------------------------------
```

```
//-----------------------------------------------------------------------------------------
// bind.
//
// Descriptinon:
//      The bind function associates a local address with a socket.
//
// Arguments:
//      s          - [in] A descriptor identifying an unbound socket.
//      name       - [in] The address to assign to the socket from the SOCKADDR structure.
//      namelen    - [in] The length of the name (unused).
//
// Return Value:
//      Success             - API_ENOERROR.
//      Socket call failed  - API_ERROR.
//-----------------------------------------------------------------------------------------
int bind(int s, const struct sockaddr* name, int namelen)
{
  union REGS inregs;
  union REGS outregs;

  inregs.h.ah     = API_BIND;
  inregs.x.bx     = s;
  inregs.x.dx     = FP_SEG(name);
  inregs.x.si     = FP_OFF(name);
  int86(TCPIPVECT, &inregs, &outregs);

  return outregs.x.dx;
} // bind.
//-----------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------
// listen.
//
// Descriptinon:
//      Place the socket in passive mode and set the number of incomimg TCP connections the
//      system will enque.
//      This call is used by a TCP server
//
// Arguments:
//      s           - [in] A descriptor identifying a bound, unconnected socket.
//      backlog     - [in] The max. number (limited to 5) of allowed outstanding connections.
//
// Return Value:
//      Success             - API_ENOERROR.
//      Socket call failed  - API_ERROR.
//-----------------------------------------------------------------------------------------
int listen(int s, int backlog)
{
  union REGS inregs;
  union REGS outregs;

  inregs.h.ah      = API_LISTEN;
  inregs.x.bx      = s;
  inregs.x.cx      = backlog;
  int86(TCPIPVECT, &inregs, &outregs);
```

```
  return outregs.x.dx;
} // listen.
//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------
// accept.
//
// Descriptinon:
//      Accept extracts the first connection on the queue of pending connections (from
//      API_LISTEN) and creates a new socket for this connection.
//      This call is used by a TCP server.
//
// Arguments:
//      sd          - [in] A descriptor identifying a socket that has been placed in a
//                    listening state with the listen function. The connection will actually
//                    be made with the socket that is returned by accept.
//      addr        - [out] An optional pointer to a buffer that receives the address of the
//                    connecting entity, as known to the communications layer. The exact
//                    format of the addr parameter is determined by the address family
//                    established when the socket was created.
//      addrlen     - [out] An optional pointer to an integer that contains the length of the
//                    address addr.
//
// Return Value:
//      Success             - New socket descriptor for the connection.
//      Socket call failed  - API_ERROR.
//-------------------------------------------------------------------------------------------
int accept(int s, struct sockaddr* addr, int* addrlen)
{
  union REGS inregs;
  union REGS outregs;

  inregs.h.ah     = API_ACCEPT;
  inregs.x.bx     = s;
  inregs.x.dx     = FP_SEG(addr);
  inregs.x.si     = FP_OFF(addr);
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx == (unsigned int)API_ERROR) {

    return API_ERROR;
  }

  *addrlen = sizeof(addr);
  return outregs.x.ax;
} // accept.
//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------
// connect.
//
// Descriptinon:
//      TCP only, The connect call attempts to make a connection to another socket (either
//      local or remote). This call is used by a TCP client.
//
// Arguments:
//      s           - [in] A descriptor identifying an unconnected socket.
```

```
//      name        - [in] The name of the socket to connect to.
//      namelen     - [in] The length of the name parameter (unused).
//
// Return Value:
//      Success             - API_ENOERROR.
//      Socket call failed  - API_ERROR.
//-------------------------------------------------------------------------------------------
int connect(int s, const struct sockaddr* name, int namelen)
{
  union REGS inregs;
  union REGS outregs;

  inregs.h.ah     = API_CONNECT;
  inregs.x.bx     = s;
  inregs.x.dx     = FP_SEG(name);
  inregs.x.si     = FP_OFF(name);
  int86(TCPIPVECT, &inregs, &outregs);

  return outregs.x.dx;
} // connect.
//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------
// send.
//
// Descriptinon:
//      TCP only, transmit message to another transport end-point send may be used only, if
//      the socket is in a connected state.
//
// Arguments:
//      s           - [in] A descriptor identifying a connected socket.
//      buf         - [in] A buffer containing the data to be transmitted..
//      len         - [in] The length of the data in buf.
//      flags       - [in] An indicator specifying the way in which the call is made
//                      --> always zero.
//
// Return Value:
//      Success             - Number of send bytes.
//      Socket call failed  - API_ERROR.
//-------------------------------------------------------------------------------------------
int send(int s, char* buf, int len, int flags)
{
  struct send_params S;
  union REGS inregs;
  union REGS outregs;

  // Fill the struct send_param S.
  S.bufferPtr      =   buf;
  S.bufferLength   =   len;
  S.flags          =   flags;
  S.toPtr          =   NULL;
  S.tolengthPtr    =   NULL;

  // API-Call.
  inregs.h.ah      = API_SEND;
  inregs.x.bx      = s;
  inregs.x.dx      = FP_SEG(&S);
```

```
  inregs.x.si      = FP_OFF(&S);
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx == (unsigned int)API_ERROR) {

    return API_ERROR;
  }

  return outregs.x.ax;
} // send.
//-----------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------
// recv.
//
// Descriptinon:
//      TCP only, receive message from another socket recv may be used only, if the socket is
//      in a connected state.
//
// Arguments:
//      s           - [in] A descriptor identifying a connected socket.
//      buf         - [out] A buffer for the incoming data.
//      len         - [in] The length of buf.
//      flags       - [in] A flag specifying the way in which the call is made:
//                    Blocking or  dontwait.
//      timeout     - [in] Timeout milliseconds.
//
// Return Value:
//      Success             - Number of received bytes (0 bytes -> timeout).
//      Socket call failed  - API_ERROR.
//-----------------------------------------------------------------------------------------
int recv(int s, char* buf, int len, int flags)
//int recv(int s, char* buf, int len, int flags, unsigned long timeout)
{
  struct recv_params R;
  union REGS inregs;
  union REGS outregs;

  // Fill the struct recv_param R.
  R.bufferPtr      =   buf;
  R.bufferLength   =   len;
  R.flags          =   flags;
  R.timeout        =   5000L; //timeout;
  R.fromPtr        =   NULL;
  R.fromlengthPtr  =   NULL;

  // API-Call.
  inregs.h.ah      = API_RECV;
  inregs.x.bx      = s;
  inregs.x.dx      = FP_SEG(&R);
  inregs.x.si      = FP_OFF(&R);
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx == (unsigned int)API_ERROR) {

    return API_ERROR;
  }
```

```
  return outregs.x.ax;
} // recv.
//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------
// sendto.
//
// Descriptinon:
//      UDP only, transmit message to another transport end-point.
//
// Arguments:
//      s           - [in] A descriptor identifying a (possibly connected) socket.
//      buf         - [in] A buffer containing the data to be transmitted.
//      len         - [in] The length of the data in buf.
//      flags       - [in] An indicator specifying the way in which the call is made
//                         --> always zero.
//      to          - [in] An optional pointer to the address of the target socket.
//      tolen       - [in] The size of the address in to.
//
// Return Value:
//      Success             - Number of send bytes.
//      Socket call failed. - API_ERROR.
//-------------------------------------------------------------------------------------------
int sendto(int s, const char* buf, int len, int flags, const struct sockaddr* to, int tolen)
{
  struct send_params S;
  union REGS inregs;
  union REGS outregs;

  // Init the struct send_param S for API-Call.
  tolen            = sizeof(struct sockaddr_in);
  S.bufferPtr      = buf;
  S.bufferLength   = len;
  S.flags          = flags;
  S.toPtr          = (struct sockaddr*)to;
  S.tolengthPtr    = &tolen;

  // API-Call.
  inregs.h.ah      = API_SENDTO;
  inregs.x.bx      = s;
  inregs.x.dx      = FP_SEG(&S);
  inregs.x.si      = FP_OFF(&S);
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx == (unsigned int)API_ERROR) {

    return API_ERROR;
  }

  return outregs.x.ax;
} // sendto.
//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------
// recvfrom.
//
```

```
// Descriptinon:
//      UDP only, receive message from another socket.
//
// Arguments:
//      s        - [in] A descriptor identifying a bound socket.
//      buf      - [out] A buffer for the incoming data.
//      len      - [in] The length of buf.
//      flags    - [in] An indicator specifying the way in which the call is made:
//                    Blocking or dontwait.
//      from     - [out] An optional pointer to a buffer that will hold the source address
upon return.
//      fromlen  - [in/out] An optional pointer to the size of the from buffer.
//      timeout  - [in] Timeout milliseconds.
//
// Return Value:
//      Success             - Number of received bytes (0 bytes -> timeout).
//      Socket call failed. - API_ERROR.
//-------------------------------------------------------------------------------------------
int recvfrom(int s, char* buf, int len, int flags, struct sockaddr* from, int* fromlen)
//int recvfrom(int s, char* buf, int len, int flags, struct sockaddr* from, int* fromlen,
unsigned long timeout)
{
  struct recv_params R;
  union REGS inregs;
  union REGS outregs;

  // Fill the struct recv_param R.
  fromlen          =   sizeof(struct sockaddr_in);
  R.bufferPtr      =   buf;
  R.bufferLength   =   len;
  R.flags          =   flags;
  R.fromPtr        =   (struct sockaddr*)from;
  R.fromlengthPtr  =   &fromlen;
  R.timeout        =   5000L; //timeout;            // milliseconds

  // API-Call.
  inregs.h.ah      = API_RECVFROM;
  inregs.x.bx      = s;
  inregs.x.dx      = FP_SEG(&R);
  inregs.x.si      = FP_OFF(&R);
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx == (unsigned int)API_ERROR) {

    return API_ERROR;
  }

  return outregs.x.ax;
} // recvfrom.
//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------
// htons.
//
// Descriptinon:
//      Converts a short value from host byte order to network byte order needed to convert
//      portnumbers.
//
```

```
// Arguments:
//      hostshort  - [in] A 16-bit number in host byte order.
//
// Return Value:
//      Success  - converted value.
//-------------------------------------------------------------------------------------------
unsigned int htons(unsigned int hostshort)
{
  union REGS inregs;
  union REGS outregs;

  // set your listening port
  inregs.h.ah      = API_HTONS;
  inregs.x.bx      = hostshort;
  int86(TCPIPVECT, &inregs, &outregs);

  return outregs.x.ax;
} // htons.
//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------
// inet_addr.
//
// Descriptinon:
//      Converts a dotted decimal IP-String to an unsigned long.
//
// Arguments:
//      cp        - [in] A null-terminated character string representing a number
//                  expressed in the Internet standard ".'' (dotted) notation
//
// Return Value:
//      Success             - An unsigned long value containing a suitable binary
//                            representation of the Internet address given.
//      Socket call failed  - API_ERROR.
//-------------------------------------------------------------------------------------------
unsigned long inet_addr(const char* cp)
{
  union REGS inregs;
  union REGS outregs;
  struct SREGS sregs;
  unsigned long lIPAddress;

  inregs.h.ah      = API_INETADDR;
  inregs.x.bx      = FP_SEG(cp);
  inregs.x.si      = FP_OFF(cp);
  sregs.es         = FP_SEG(lIPAddress);
  inregs.x.di      = FP_OFF(lIPAddress);
  int86x(TCPIPVECT, &inregs, &outregs, &sregs);


  if(outregs.x.dx == (unsigned int)API_ERROR) {

    return outregs.x.dx;
  }

  return lIPAddress;
} // inet_addr.
```

```
//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------
// inet_ntoa.
//
// Descriptinon:
//      Converts an unsigned long IP address to a dotted decimal IP string
//
// Arguments:
//      in          - [in] Pointer to the ip adress.
//      out         - [out] Pointer to the string buffer, where this function can fill in
//                    the converted value.
//                    The buffer must have the length of 17 Bytes!
//
// Return Value:
//      Success  - API_ENOERROR.
//-------------------------------------------------------------------------------------------
int inet_ntoa(unsigned long* in, char* out)
{
  union REGS inregs;
  union REGS outregs;
  struct SREGS sregs;

  inregs.h.ah     = API_INETTOASCII;
  inregs.x.bx     = FP_SEG(in);
  inregs.x.si     = FP_OFF(in);
  sregs.es        = FP_SEG(out);
  inregs.x.di     = FP_OFF(out);
  int86x(TCPIPVECT, &inregs, &outregs, &sregs);

  return outregs.x.dx;
}  // inet_ntoa.
//-------------------------------------------------------------------------------------------


// special socket functions
//

//-------------------------------------------------------------------------------------------
// TCPIP_Sleep.
//
// Descriptinon:
//      The application sleeps for given milliseconds.
//
// Arguments:
//      howlong  - milliseconds.
//
// Return Value:
//      Success  - API_ENOERROR.
//-------------------------------------------------------------------------------------------
void TCPIP_Sleep(unsigned int howlong)
{
  union  REGS  inregs;
  union  REGS  outregs;

  inregs.h.ah     = API_SLEEP;
  inregs.x.bx     = howlong;
```

```
  int86(TCPIPVECT, &inregs, &outregs);
} // TCPIP_Sleep.
//-----------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------
// TCPIP_GetWaitingBytes.
//
// Descriptinon:
//      Get the number of bytes on a socket, waiting for read.
//
// Arguments:
//      sd          - Socketdescriptor.
//      error       - .
//
// Return Value:
//      Success             - The number of bytes, waiting for read.
//      Socket call failed  - API_ERROR.
//-----------------------------------------------------------------------------------------
int TCPIP_GetWaitingBytes(int sd, int* error)
{
  union REGS inregs;
  union REGS outregs;

  *error = API_ENOERROR;

  inregs.h.ah      = API_GETRCV_BYTES;
  inregs.x.bx      = sd;
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx== (unsigned int)API_ERROR) {

    *error = outregs.x.ax;
    return API_ERROR;
  }

  return outregs.x.ax;
} // TCPIP_GetWaitingBytes.
//-----------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------
// TCPIP_ResetConnection.
//
// Descriptinon:
//      Abort a connection on a socket. Works only with TCP sockets!
//
// Arguments:
//      sd          - Socketdescriptor.
//      error       - .
//
// Return Value:
//      Success             - API_ENOERROR.
//      Socket call failed  - API_ERROR.
//-----------------------------------------------------------------------------------------
int TCPIP_ResetConnection(int sd, int* error)
{
  union REGS inregs;
  union REGS outregs;
```

```
  *error = API_ENOERROR;

  inregs.h.ah      = API_RESETCONNECTION;
  inregs.x.bx      = sd;
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx == (unsigned int)API_ERROR) {

    *error = outregs.x.ax;
  }

  return outregs.x.dx;
} // TCPIP_ResetConnection.
//-----------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------
// TCPIP_SetLinger.
//
// Descriptinon:
//      Set linger time on close. Works only with TCP sockets!
//
// Arguments:
//      sd          - Socketdescriptor.
//      seconds     - Linger time in seconds, 0 means linger off.
//      error       - .
//
// Return Value:
//      Success             - API_ENOERROR.
//      Socket call failed  - API_ERROR.
//-----------------------------------------------------------------------------------------
int TCPIP_SetLinger(int sd, int seconds, int* error)
{
  union REGS inregs;
  union REGS outregs;

  *error = API_ENOERROR;

  inregs.h.ah = API_SETLINGER;
  inregs.x.bx = sd;
  inregs.x.cx = seconds;
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx == (unsigned int)API_ERROR) {

    *error = outregs.x.ax;
  }

  return outregs.x.dx;
} // TCPIP_SetLinger.
//-----------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------
// TCPIP_SetReuse.
//
// Descriptinon:
//      Works only with TCP sockets! This is necessary, if a listening socket was closed and
```

```
//      will be open and bind on the same port as it was bind before.
//
// Arguments:
//      sd          - Socketdescriptor.
//      error       - .
//
// Return Value:
//      Success             - API_ENOERROR.
//      Socket call failed  - API_ERROR.
//-----------------------------------------------------------------------------------------
int TCPIP_SetReuse(int sd,int* error)
{
  union REGS inregs;
  union REGS outregs;

  *error = API_ENOERROR;

  inregs.h.ah = API_SETREUSE;
  inregs.x.bx = sd;
  int86(TCPIPVECT, &inregs, &outregs);

  if(outregs.x.dx == (unsigned int)API_ERROR) {

    *error = outregs.x.ax;
  }

  return outregs.x.dx;
} // TCPIP_SetReuse.
//-----------------------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------
// end tcpip.c
```