JABBA Embedded Webserver

Lukas Karrer, Thomas Moser, Thomas Zwicker

Entwicklungsbericht

ABSTRACT

Devices like balances or coffee-machines are going online in near future. Designed for point-to-point communication or no interface at all, special innovation is needed to use todays Internet technology with these appliances.

The JABBA concept provides a solution, which enables fast and easy development, while keeping maximum flexibility. Using an adaptable software layer on the interface, JABBA overcomes the restrictions imposed by network and high level protocols which are commonly used today. Resource-hungry applications are shifted to the desktop PC using JAVA technology. JABBA's layered software-design approach splits up development in low-level C++ firmware and easy JAVA programming for control and GUI.

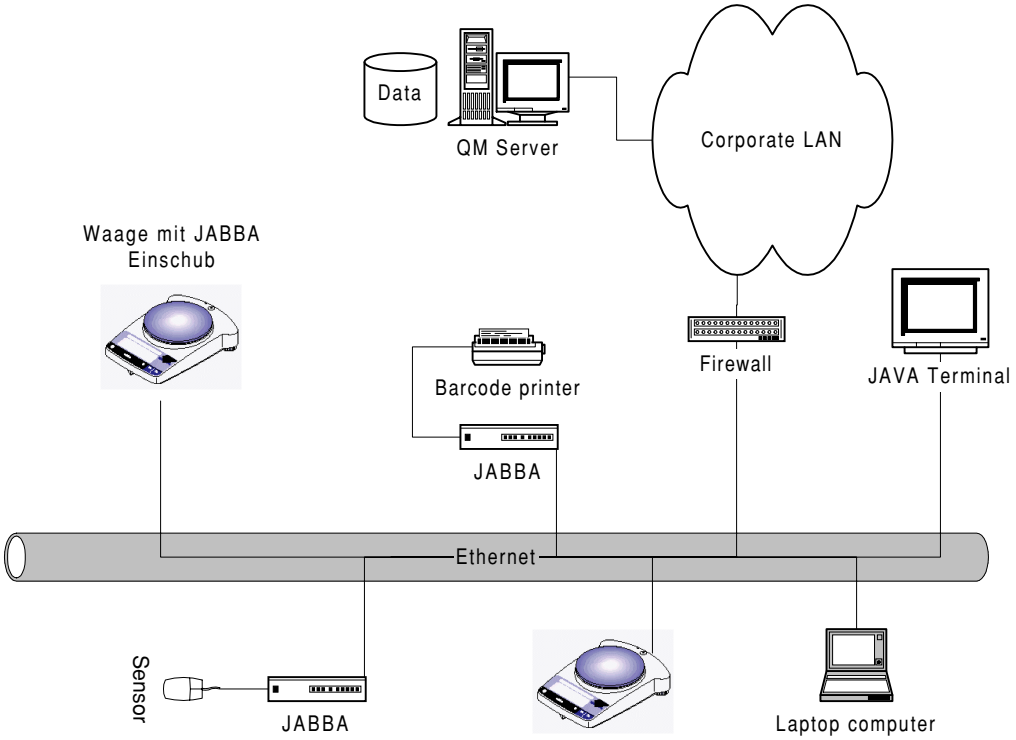
Inhaltsverzeichnis

1.0 Aufgabenstellung

Die Aufgabenstellung gemäss unseres Definitionspapieres (siehe Anhang A) lautete: "Implementierung eines Geräte - Internet Interfaces mit möglichst geringen Hardwareansprüchen". In der ersten Phase der Arbeit setzten wir uns weitere Randbedingungen. Wir wollten zusätzlich eine einfache Programmierbarkeit erreichen, Jabba sollte einfach und nur mit wenig Hintergrundwissen an eine neue Applikation angepasst werden können. Es war nie das Ziel, ein fertiges Produkt, resp. "the world's cheapest Web server" zu entwickeln, dazu müsste die Hardware höher integriert werden.

2.0 Voraussetzungen

Wägeapplikationen vom MT werden in industriellen Umgebungen eingesetzt, wobei Prozesswaagen meist nicht einmal ein Display haben. Es müssen Waagen, Barcode Printer, Qualitiy Management Server, Konfigurationsterminals und mehr miteinander kommunizieren. Die Vernetzung ist also ein hochrangiges Problem. Bisher wurde v.a. ein CAN - Bus dazu verwendet. In den letzten Jahren fand ein Vordringen der Ethernet - Technologie von der Prozessleitebene immer näher zum eingebetteten System statt. Parallel dazu nahm die globale Vernetzung rasant zu. Um den CAN - Bus anzupassen, werden CAN - Ethernet Interfaces verwendet. Eine weitere Möglichkeit, der steigenden Vernetzung Rechnung zu tragen besteht darin, das Ethernet direkt an das eingebettete System zu bringen. In dieser Richtung war bei Mettler Toledo, kurz MT, bis anhin noch nicht viel passiert. Entsprechend waren wir hier sehr frei, es hiess so etwa: "macht mal was". Es sollte einfach billig und flexibel sein.



Auch bei der Hardware hatten wir keine Vorschriften zu beachten. Uns wurde jedoch klar, dass wir nur dann vom KnowHow und den vorangegangenen Arbeiten von Mettler profitieren konnten, wenn wir ihre Hardware und Entwicklungsumgebung übernehmen würden. Dies war auch in unserem Sinn, da wir uns so auf den "proof of concept" von Jabba konzentrieren konnten.

2.1 Übernommene Teile

Demzufolge haben wir folgendes übernommen:

- Tigris - Entwicklungsplattform (OSE - RTOS, Compiler, Debugger)
- Tigris - Hardwarekomponenten (68000, NIC)
- Portierter Waterloo TCP / IP Stack (nach Test)

3.0 Eigener Lösungsansatz

Der Lösungsansatz JABBA umfasst drei Kernkonzepte:

1. Verlagerung der Intelligenz vom Embedded Device in einen PC
2. Konsequent geschichteter Aufbau
3. Device-Abhängiger Softwarelayer

Damit erreichen wir eine einfache Erweiterbarkeit und Anpassungsfähigkeit an neue Geräte.

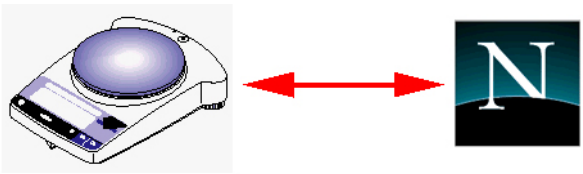
Im folgenden ist jedes dieser Kernkonzepte erklärt.

3.1 Verlagerte Intelligenz

Embedded devices sind in Bezug auf Speicherkapazität und Leistung limitiert. Kosten, Stromverbrauch und vorhandener Raum begrenzen den Ausbau dieser Parameter. Das Interface zum Menschen muss jedoch meistens luxuriös ausgestattet sein. Die genannten Anforderungen laufen somit in die entgegengesetzte Richtung.

JABBA verlagert die Intelligenz dorthin, wo sie ausreichend vorhanden ist - in einen Personal Computer. Komplexe Anwendungen und elaborierte GUI's werden vom embedded device getrennt.

Der Anwender lädt mit seinem WebBrowser ein auf der JABBA-Box gespeichertes JAVA Applet. Hierfür ist ein vollständiger WebServer implementiert. Das JAVA Applet kommuniziert nun mit dem embedded Device mit einem einfachen Protokoll. So kann das angeschlossene Gerät gesteuert oder Werte abgefragt werden.



Die Programmiersprache JAVA ermöglicht eine platformunabhängige Entwicklung der Bedienerapplikation. Weiter sind für JAVA mächtige Bibliotheken vorhanden, die das programmieren von graphischen Oberflächen sehr vereinfachen.

Neben JAVA Applets kann der Webserver beliebige andere Dateien verteilen. So kann beispielsweise eine Online-Hilfe in HTML oder Dokumentation in PDF auf dem Device verfügbar gemacht werden.

3.2 Geschichteter Aufbau

Die Programmierung einer Applikation zur Steuerung eines Gerätes umfasst viel KnowHow. Wissen über die GUI- und Netzwerkprogrammierung sowie genaue Kenntnisse über die technischen Eigenschaften des angeschlossenen Gerätes sind nötig. Die

konsequente Schichtung des Softwareaufbaus ermöglicht eine Trennung der Entwicklungsaufgaben.

Der Applikationsentwickler muss sich nur um das GUI und die eigentliche Funktionalität kümmern. Eine Softwarebibliothek (API) stellt dem Programmierer einfache Primitiven zur Verfügung, über die mit dem angeschlossenen Gerät interagiert werden kann. In unserer Implementation stellt die JAVA Bibliothek beispielsweise eine Methode GetWeight() zur Verfügung, die das aktuelle Gewicht der angeschlossenen Waage liefert. Der Applikationsentwickler muss sich demnach nur mit der Weiterverarbeitung und Darstellung der Resultate kümmern.

Das angesprochene API wird von einem Techniker unterhalten, der das angeschlossene Gerät kennt. Er weiss, welche Befehle oder Abfolge von Befehlen eine bestimmte Aktion auslösen.

Um die für jedes Gerät spezifische Bibliothek von der Netzwerkprogrammierung zu trennen, wird wiederum auf eine tieferliegende Bibliothek zugegriffen. Die Klasse JABBAhttp stellt Methoden wie “GetFromJabba()” und “SendToJabba()” zur Verfügung. Der Programmierer kann ohne Kenntnisse in Netzwerkprogrammierung oder der darunter liegenden Protokolle mit der JABBA Box kommunizieren. Diese Bibliothek wird vom Softwareingenieur der JABBA Firmware unterhalten. Die Schichtung ist nochmals in Abb. 1, “Geschichteter Aufbau des JABBA Konzepts,” auf Seite 7 verdeutlicht.

Neben der Kompetenzaufteilung hat der geschichtete Aufbau einen weiteren Vorteil. Die Entwicklung neuer Applikationen ist sehr einfach. Es muss nur die oberste Softwareschicht erneuert werden. Auch die Weiterentwicklung des Protokolls zwischen Applet und JABBA geht problemlos, ohne Änderung der Applikation - es muss nur die Bibliothek ausgewechselt werden.

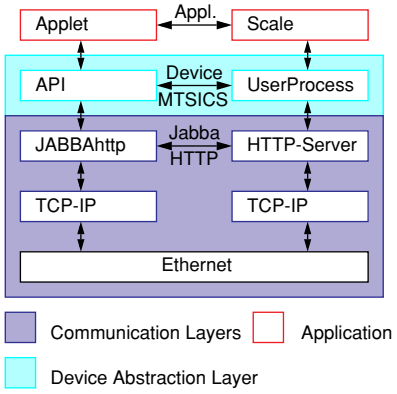


FIGURE 1. Geschichteter Aufbau des JABBA Konzepts

3.3 Device-Abhängiger Softwarelayer

Zwischen Webserver und Gerät ist eine weitere Softwareschicht eingebaut. Der sogenannte UserProcess (Siehe Abb. 1, “Geschichteter Aufbau des JABBA Konzepts,” auf Seite 7) ist ein eigener Prozess, der die Schnittstelle zum angeschlossenen Gerät bildet. In der einfachsten Form waltet der UserProcess als Gegenstück zur JABBAhttp Klasse. Die über das HTTP Protokoll verschickten Daten werden ausgepackt und an die serielle Schnittstelle weitergeleitet.

Die Mächtigkeit des UserProcesses liegt in seiner Erweiterbarkeit. Der UserProcess kann für ein bestimmtes Gerät spezialisiert werden. So werden beispielsweise zeitkritische Vorgänge unabhängig von einer Applikation oder dem Netzwerk ausführbar. Dadurch können Echtzeitprobleme von Ethernet und HTTP umgangen werden.

3.4 Polling über HTTP

JABBA wird in einer verteilten Umgebung eingesetzt. Eine starre Punkt zu Punkt Verbindung zwischen zwei Geräten, wie sie mittels serieller Verbindung existiert, ist nicht zweckmässig. Um den Mehrfachzugriff auf ein Device zu ermöglichen, wird vom Applet aus gepollt. Damit ergeben sich mit einem einfachen UserProcess keine dauernd offenen TCP-Verbindungen.

Die Kommunikation zwischen Browser und JABBA resp. Applet und embedded Device benutzt das HTTP Protokoll. So kann problemlos über mehrere Netzwerke und Firewalls hinweg mit einer JABBA-Box Kontakt aufgenommen werden.

Allfällige Nachteile, die durch diese Strategie anfallen, können durch geeignete Erweiterung des UserProcesses umgangen werden.

4.0 Resultate

Die Arbeit bei Mettler hat uns gezeigt, dass die JABBA-Idee eine gute Lösung für unser Problem darstellt. Die einfache Applet-Entwicklung mit der MTBalance-Klasse zeigte ihre Vorteile als es darum ging, die Webseiten inklusive Applets von Mettler-Leuten zu gestalten. Durch das MTBalance API konnten wir diese Schnittstelle abgeben und dann darunter weiterentwickeln ohne auf die Applets rücksicht zu nehmen.

Der UserProcess wurde leider noch nicht verändert, hier haben wir selber aber bei den Probeuploads immer mehr Freude an der Idee gekriegt. Abändern, compilieren & linken und dann uploaden und laufenlassen ist in einigen Minuten möglich. Anpassungen an ein neues Device sind richtig bequem zu machen.

Die Konfiguration und Systemfunktionen konnten wir am 25 Jahre TIK-Jubiläum selber unter realen Bedingungen testen. JABBA wurde da an das ETH-Netz angeschlossen. Die IP-Konfiguration war über die serielle Schnittstelle machte keine Schwierigkeiten. Und danach konnten wir auch von den Suns auf JABBA zugreifen.

Rückblickend waren wir bei den Tests ab und zu erstaunt was JABBA eigentlich alles konnte. Von der kleinen Hardware über den TCP/ IP-Stack bis zu Server und Applets sind alle nötigen Funktionen sauber implementiert.

Das Polling-Konzept bewährte sich zwar für diese Arbeit, es muss aber bei komplexeren Aufgaben durch eine Client-Verwaltung im UserProcess abgelöst werden. Hier haben wir keine gute Lösung gefunden um diesen Programmieraufwand für den UserProcess zu verkleinern.

Der “proof of concept” ist uns gelungen.

5.0 Rückblick

Unsere Semesterarbeit umfasst ein weites Spektrum. Technologien wie JAVA, HW-Design, OOP, low-level TCP/IP und höher liegende Protokolle kamen zum Einsatz. Obschon jeder von uns sein Spezialgebiet hat, konnten wir unseren Horizont enorm erweitern.

Neben den technologischen Aspekten kamen auch die ‘menschlichen’ dazu. Für uns war es die erste Projektbezogene Gruppenarbeit. Obschon viel Zeit für die Synchronisation der Arbeit aufgewendet werden musste, waren die Resultate der gemeinsamen Sitzungen immer positiv. Durch das argumentieren eigener Ansätze fanden wir gemeinsam Lösungen, die Stand halten.

Auch das industrielle Umfeld war eine neue Erfahrung. Anfangs mussten wir einiges an Überzeugungsaufwand für das JABBA Konzept leisten. Skeptische Voten von Seiten der Firma Mettler Toledo zeigten uns jedoch auch Schwachstellen auf.

Die seriöse Arbeitsplanung (Abb. , “,” auf Seite 37), die wir noch vor Beginn der Arbeit erstellt haben erleichterte uns gewisse Entscheidungen. Fortwährend konnte der Ablauf der Arbeit kontrolliert werden. So gerieten wir nie in Gefahr, uns in Details zu verlaufen. Die vorgegebenen Milestones “Networking”, “Minimalrealisation” und “Endprodukt” konnten ziemlich genau eingehalten werden. Einzig die eingeplante Zeit für Dokumentation reichte nicht aus.

Während der ganzen Arbeit, reservierten wir den Montag Nachmittag für Strategiesitzunegen. An diesem Halbtag rapportierten wir gegenseitig über unseren Fortschritt und besprachen allfällige Probleme. Gemeinsam wurde für anstehende Fragen und Probleme Lösungsmöglichkeiten gesucht.

An dieser Stelle möchten wir uns bei folgenden Personen für die Mithilfe bedanken:

- Beat Gamper für die Layouts und den Einsatz
- Tarik Oelmez für die Hints
- Werner Langenegger fürs Probelesen der Doku
- Thomas Hochueli für die Unterstützung bei TCP-Problemen
- Jonas Greutert für die Betreuung von Seite ETHZ
- Marcel Walder fürs Annehmen des Projekts

6.0 Arbeitsbericht: Hardware

6.1 Vorgaben

Mit dem Entschluss bei Mettler Toledo diese Arbeit durchzuführen stand auch gleich die Hardwareplattform fest. Dies ist ein Motorola 68000er Prozessor mit 256k SRAM und 1MB Flashspeicher sowie Peripherie.

Um unserem Ziel einer billigen Hardware nahezukommen, schien dies eine gute Variante, da Mettler hohe Stückzahlen dieser Komponenten braucht. Leider trifft das genau beim Ethernet-Chip nicht zu, was den Preis am Ende stark in die Höhe trieb.

Ausserdem standen uns bei Mettler etliche Entwicklungstools zur Verfügung.

6.2 Ziel

Ausgehend von der bestehenden ProdBox, einer Testbox mit viel Peripherie, wollten wir eine Hardware der Grösse eines Mars-Schokoriegels entwickeln.

Neu benötigten wir anstatt 256kB RAM vor allem zwei unabhängige Flash-Speicher und nur noch die Ethernet- und serielle Schnittstellen.

6.2.1 Design

Es entstand so eine Variante der ProdBox von Mettler mit folgenden Merkmalen:

- 2*1MB Flash
- 256kB RAM, aufgeteilt in 2*128kB High- und Low-Byte
- Ethernet, ein Kontroller sowie Transceiver
- RS232, integriert in den Basco-ASIC
- 20MHz Motorola 68000er CPU

Dabei entfallen Bausteine von der ProdBox wie RealTimeClock, I2C-EE-ROM, I2C-Treiber, externe UARTS und das batteriegepufferte RAM.

6.2.2 Kosten

Die Bauteilkosten belaufen sich somit auf ca. Fr.40.-, wobei der NIC mit Fr. 18.- den Preis im Wesentlichen bestimmt.

6.3 Fragestellungen

6.3.1 Daten/ Programm-Aufteilung im Flash:

Die Server-Software soll den Ethernet-Datenupload ermöglichen, wird aber selber im Flash ausgeführt. Somit wäre ein Umladen des Codes ins RAM während dem Datenupload nötig, da der Schreibvorgang im Flash nicht durch Leseanforderungen der CPU

unterbrochen werden darf. Ausserdem wäre der im Flash verbleibende Speicherplatz klein.

Ein zweiter Flashbaustein schafft bei beiden Problemen Abhilfe. Ein Flash wird für den Programmcode verwendet, das Andere für die Upload-Daten.

6.3.2 Upload der Software

Mettler verwendet auf ihrer ProdBox Kontaktierpunkte mit Hilfe derer das Flash im eingebauten Zustand geladen werden kann. Diese Pads müssen auf unserem PCB entfallen.

Das Nachbarwerk in Nänikon konnte uns einen Flashprogrammer ausleihen um die Flash-Chips vor dem Löten mit dem Bootcode zu programmieren.

6.3.3 Default-Werte und Absturz-Absicherung

Um einen eventuell abstürzenden UserProcess wieder löschen zu können und auch um Default-Konfigurationswerte setzen zu können, haben wir uns für Lötpads auf der Hardware-Oberseite entschieden. Platzsparend und leicht mit Jumper bestückbar sind sie für die Entwicklung nötig und für den Gebrauch unsichtbar.

6.3.4 Address-Mapping für das neue Flash

Da die Chipselect-Signale im Basco ASIC generiert werden musste ich das dort implementierte Mapping übernehmen. Glücklicherweise konnte ich einfach das CS-Signal des vorigen NV-RAMs übernehmen.

6.3.5 Serielle Schnittstelle

Der im Basco integrierte UART wird schon bei der ProdBox als Haupt-UART verwendet und schafft Baudraten bis 115kbps. Allerdings entfällt auf unserem Einschub der MAX232 zur 12V-Pegelgeneration, da der Duploeinschub mit TTL-Signalen arbeitet.

Dafür haben wir einen Zusatzprint gemacht mit einem MAX232 und einem 9Pol DSUB-Stecker um Computer anschliessen zu können.

6.3.6 Abmessungen

Um die geforderten Abmesssungen einzuhalten (Duplo-Slot) konnten wir nicht alles auf einen Print nehmen. Wir entschieden uns für einen Sandwich-Aufbau mit je einer doppelseitigen Printplatte.

Nach einem Probe-Placing der Bauteile war die Machbarkeit klar. Allerdings war schon dann abzusehen dass der Print sehr gut ausgenutzt wird (4 Layer).

6.4 Layout/ Bestückung

Das Layout wurde von Beat Gamper angefertigt. Er bestückte auch die ersten Prototypen.

Es wurden durchwegs SMT-Komponenten verwendet. ESD-Schutzmassnahmen wurden soweit möglich beachtet (Leitende Schaumstoffe bei Transport, Armband beim Löten).

6.5 Inbetriebnahme

Nach sanftem Hochdrehen der Strombegrenzung schien alles ok. Sogar der Flashloader über die serielle Schnittstelle funktionierte. Wenigstens nach dem Umlöten des MAX202 auf dem Zusatzprint.

Soweit schien alles ok. Nach dem Aufspielen der Jabba-Software kam dann allerdings das grosse Problem: die ARP-Requests wurden nicht beantwortet.

6.5.1 Fehlersuche

Da die Software auf der ProdBox funktionierte, das Programm auf Jabba aber sonst tadellos ausgeführt wurde, stand ich vor einem grossen Fragezeichen. Dies konnten wir durch das Pingen des PCs von Jabba aus feststellen. Danach liefen die Browserzugriffe solange der ARP-Cache des PCs nicht gelöscht wurde. Folglich untersuchte ich den Eingang der ARP-Pakete.

Mögliche Ursachen:

1. Software benötigt nichtexistierende Hardware für ARP
2. Ethernet-Controller erhält falsche Daten
3. Ethernet-Controller übermittelt falsche Daten
4. Konfiguration falsch
5. neue Chiprevisionen
6. Timingprobleme

Punkt 1 konnte ich schnell durch eine Nachkontrolle ausschliessen.

Der 2. Punkt war mit Hilfe des Network-Surveyors und durch die Annahme des ARP-Pakets durch den Chip (CRC-Check) ebenfalls bald auszuschliessen.

Beim 3. Punkt konnte ich feststellen dass der Chip ein Datum, genau das 6.Byte nur der ARP-Pakete, ausliess beim Einlesen in den Speicher. Dass er aber ein vollständiges Paket erhielt konnte ich aus der Interruptmeldung an die CPU schliessen. Somit musste im Datentransfer vom NIC in den Speicher ein Fehler auftreten. Mit Hilfe eines Memorydumps zeigte sich jenes ausgelassene 6. Byte. Die Einleseroutine benützte die AutoIncrement-Funktion des NIC. Probeweise ersetzte ich diese durch ein Step-by-Step auslesen. Dies löste das Problem der ARP Pakete.

Punkt 4 & 5 kontrollierte ich zur Sicherheit noch. Die Konfiguration als auch ID-Daten stimmten mit der ProdBox überein.

Punkt 6 war unwahrscheinlich da sich diesbezüglich nichts geändert hat. Allerdings verifizierte ich das Design nochmals anhand der Datenblätter.

Damit aber noch nicht genug. Später tauchte als nächstes Problem der Upload von Binärdaten auf. Beim Download der HTML-Daten stürzte die neue Hardware ab sobald die Daten nicht mehr aus HTML-Text bestanden sondern aus Bildern oder Applets. Auch dieser Fehler trat nur mit der Jabba-Hardware auf.

Dieses Problem schien demzufolge die gleiche Ursache wie das vorige zu haben. Erstaunlicherweise war das ganze Datenabhängig und lief einwandfrei auf der ProdBox. Implizierte das eine ein Softwareproblem, so schloss das andere es aus. Um Timingprobleme auszuschliessen, verschob ich die fehlerauslösenden Daten im Upload. So konnte ich feststellen dass 300kB Text durchkamen, der Fehler aber genau an der Stelle auftrat, wo ich zusätzlich Binärdaten einfügte.

Um sicherzugehen dass der alte Fehler nicht den neuen verursacht, checkten wir auf Optimierungsprobleme im NIC-Treiber. Tatsächlich wurde der Fehler damit behoben. Die Reihenfolge der Chipzugriffe und auch die Variablen zum Auslesen werden nun durch die VOLATILE-Angabe gesichert. So löste sich dieses sehr hartnäckige Problem.

6.6 Fazit

Das Ziel den Webserver unter Fr.100.- verkaufen zu können dürften wir nicht erreicht haben, liegen doch die Kosten des Ethernet-Chips alleine bei Fr.18.-.

Die Vollkostenrechnung von Marcel Walder, dem Projekzuständigen, ergab einen Preis von 80$ bis 120$ je nach Stückzahl. Wollte man unbedingt darunter kommen, so könnte man für grösste Stückzahlen das Ganze in Silizium implementieren. Doch da liegen die grossen Konzerne vorn. Daher zielt unser Projekt vor allem auf leichte Programmierbarkeit ab, um schnell angepasst zu sein.

7.0 Arbeitsbericht: Jabba HTTP Server

7.1 Exploration

Die Auswahl an implementierbaren Protokollen zwischen Browser und Jabba Device ist relativ klein. Möglich sind die definierten Standards, HTTP Version 1.0 und Version 1.1. Als dritte Variante stand eine Implementation eines Subsets der Version 1.1 zur diskusion.

Mehr Freiheit bieten die Zusätze, die neben der Implementation des eigentlichen Protokolls erforderlich sind. Ich denke an das Filesystem, wo die Dateien abgelegt werden oder an die Implementation und Einbettung der HTTP Authentifikation.

7.2 Entscheidungen

Die Auswahl an Implementationen des HTTP Protokolls ist sehr gross. Aus diesem Grund lag die Idee nahe, eine Implementation zu portieren. Wir merkten jedoch relativ schnell, dass keine der von uns betrachteten HTTP Server die besonderen Anforderungen und Bedürfnisse von Embedded Systemen abdecken konnten. Die meisten offenen Server sind für das Betriebssystem Linux geschrieben. Neben dem Linux Kernel steht eine solides Netzwerk API (Berkley Sockets) und ein Filesystem zur Verfügung. Dies mit einer Vielfallt von Funktionalität, die in embedded systems weder brauchbar noch implementierbar sind. Weiter fanden wir keine Implementation, die für den benötigten Arbeitspeicher optimiert waren. Die für den Webserver verfügbaren 128k RAM auf JABBA reichten mit vorhandenen Implementationen nirgends hin, insbesondere bei Uploads von grösseren Datenmengen via der HTTP Post operation.

Aus diesen Gründen beschlossen wir, den Webserver vollständig neu zu schreiben.

7.2.1 JABBA Webserver

Der HTTP Server auf JABBA wurde nach RFC 1945 für die HTTP Version 1.0 aufgebaut. Implementiert sind die GET und POST Methoden. Jede Anfrage an den Server wird vom JabbaSrv Task in einer Datenstruktur abgelegt. Die Datenstruktur, die sowohl die vom IP-Stack verwaltete Waterloo Socket-Struktur wie auch eine eigene Struktur für die Variablen des Webservers umfasst, wird durch die einzelnen Funktionen des Webservers weitergereicht.

Wir fassten den Grundsatzentscheid, dass jede Anfrage sequentiell abgearbeitet wird. Eine neue Anfrage wird erst bearbeitet, wenn die alte vollständig verarbeitet und die Antwort dem Client zurückgegeben wurde.

Die Rechtfertigung, jeder Anfrage eine neue Datenstruktur zu geben liegt in der Erweiterbarkeit des Webservers. Das JABBA Konzept, das auf polling basiert, ist relativ ineffizient mit HTTP 1.0. Die Möglichkeit des HTTP 1.1 Protokolls, über eine TCP-Verbindung mehrere Request zu bearbeiten, haben wir für das schnelle Polling von Daten ins Auge gefasst. Die dem Webserver zugrundeliegenden Datenstrukturen sind so ausgelegt, dass ohne Umstellung auf die neuere Version des HTTP Protokolls gewechselt werden könnte. Die Implementation von HTTP Version 1.1 lag jedoch aus Zeitgründen nicht im Rahmen dieser Arbeit.

JABBA's Webserver muss mit sehr grossen Datenuploads zurecht kommen. Das im nächsten Abschnitt beschriebene Filesystem kann bis zu einem Megabyte Daten umfassen, die als monolythischer Block mittels einer HTTP POST Operation auf JABBA geladen werden. Der Webserver ist so implementiert, dass er für Datenuploads nie mehr als 2 Kilobyte grosse Datenblöcke im RAM allozieren muss. Die Flusskontrolle der Daten wird vom TCP Protokoll automatisch wahrgenommen, sodass keine Gefahr besteht, den JABBA Webserver zu 'überschwemmen'.

Im embedded Bereich ist die Stabilität zentral. Der Webserver wie auch TCP / IP muss absolut stabil und ohne Memory-Leak arbeiten. Ein JABBA-Device kann über Jahre ohne Reboot am Netz hängen. Aus diesem Grund ist bei der Konzeption des Webservers ein spezielles Augenmerk auf diese Tatsache gegangen. Timeouts von HTTP Anfragen werden sauber abgearbeitet und die Datenstrukturen neu initialisiert.

7.2.2 JABBA File System

Die embedded Umgebung von Jabba stellt andere Ansprüche an das Filesystem, als ein ausgewachsener Webserver. Auf JABBA, wo Webseiten weniger dynamisch verändert werden als dies in anderem Umfeld der Fall ist, sollte ein monolythischer Block genügen. Der Austausch einzelner Files ist somit nicht möglich. Der Webserver greift via eine einzige Funktion auf das Filesystem zu. Wird in Zukunft ein ausgereiftes Filesystem eingebaut, so ist am Webserver nur die fHttp_get_file() Funktion anzupassen.

7.2.3 JABBA Authentication

Um den Zugriff auf ein JABBA device zu kontrollieren, sind passwortgeschützte Seiten nötig. Dabei soll nicht nur Webinhalt geschützt werden, sondern auch Zugriff auf System- und Userfunctions. So kann beispielsweise eine Konfigurationsänderung nur nach Eingabe von Login und Passwort geschehen. Aus Zeitgründen umfasst der JABBA Webserver nur ein login / passwort.

Passwortschutz wurde mittels der in RFC 1945 beschriebenen Methode implementiert. So kann wiederum mittels Browser oder über ein Programm auf JABBA zugegriffen werden.

7.3 Ausblick / Zukunft

Vergleicht man die minimale Implementation des JABBA Webservers mit Produkten wie APACHE, so würde der Abschnitt über mögliche Ausblick beliebig gross :=) Insbesondere bezüglich Komfort kann beim JABBA Webserver einiges gemacht werden. Folgende Punkte sehe ich als zukünftige Ausbauschritte:

- Abstimmung der Timeouts
- Directory Index - Anzeige der Dateien in einem Ordner
- HTTP Authentication mit verschiedenen Passwörtern auf Datei-Stufe
- HTTP 1.1 Implementation mit KeepAlive
- Vom Benutzer konfigurierbare MimeTypes Tabelle

8.0 Arbeitsbericht: UserProcess

8.1 Probleme der Jabba Konzeption:

Unser Ansatz mit der Appletprogrammierung hat nicht nur Vorteile, wir handelten uns auch ein paar Probleme ein: Effizienz / Echtzeitfähigkeit: Die Steuerung eines Gerätes durch ein Java Applet oder Applikation hat gewisse Nachteile bezüglich Effizienz. So produziert Polling ein reger Netzwerkverkehr, auch ist die “Echtzeitfähigkeit” wegen des relativ hohen Netzwerkdelays eingeschränkt. Dies liegt zum Einen am Netzwerk selbst, zum Anderen am dem von uns verwendeten HTTP 1.0 - Protokoll, welches nach einer Antwort auf eine Anfrage jeweils die Verbindung schliesst. Es muss also für jeden Messwert eine eingene Verbindung aufgebaut und wieder abgebrochen werden. Anpassung an eine eigene Anwendung: Nicht jedes zu steuernde Gerät verfügt über eine standardisierte RS232 Schnittstelle. Der Programmierer soll die Möglichkeit haben, die vorhandenen I/O Möglichkeiten auszuschöpfen, oder z.B. eine Zugriffsverwaltung zu implementieren. Solche Änderungen müssen auf Jabba selbst erfolgen. Es soll aber auch hier auf einfache Weise programmierbar sein. Interruptfähigkeit: Jabba soll nicht nur durch Polling Daten preisgeben, sondern auch selbständig auf gewisse Ereignisse reagieren und gegebenenfalls das Vorgefallene weiter kommunizieren können.

8.2 Zweck des UserProcesses

Um eine Lösung für obige Probleme anzubieten, führten wir den UserProcess ein. Da er auf der Jabba - Box selber läuft, also nahe an der zu steuernden Maschine ist, ermöglicht er eine hardwarenahe Programmierung. Damit ist mit Jabba auch die “traditionelle Art” der Programmierung von eingebetteten Systemen möglich. Man kann so einige Restriktionen vom Applets wettmachen.

8.3 Warum ein eigener Prozess?

Der UserProcess fasst alle anwendungspezifischen Eigenschaften in einem einzigen Prozess zusammen. Durch diese Abkapselung ist es möglich, Jabba ohne durchgehende Systemkenntnisse den eigenen Bedürfnissen anzupassen. Man hat eine klare Schnittstelle zum Server und zur seriellen Schnittstelle, ansonsten ist man frei: Die existierenden Ein- und Ausgänge können angesteuert oder den TCP/IP Stack zur direkten Kommunikation mit dem Client verwendet werden. Einen einzelnen Prozess kann man an einem beliebigen Ort im Adressraum platzieren und laufen lassen. So können wir ihn separat, und ohne den Rest von Jabba, kompilieren und dynamisch uploaden. Nach einem einfachen Neustart ist der Code dann aktiv.

8.4 DefaultUserProcess

Wir haben einen einfachen UserProcess als Vorlage zur weiteren Programmierung und zur Demonstration von Jabba geschrieben. Er soll v.a. die IPC Mechanismen zum Server und zur Uart und den Konfigurationsparser beispielhaft zeigen. Er ist an die Waagenschnittstelle vom Mettler Toledo (MT-SICS) angepasst, aber dennoch sehr allgemein verwendbar. Der DefaultUserProcess ist zusammen mit dem Rest von Jabba kompiliert

und gelinkt. Er ist also immer vorhanden, wird aber beim Hochfahren bei kurzgeschlossenem Jumper 2 durch den veränderten UserProcess im Datenflash ersetzt. So kann man bei einem fehlerhaften, eigenen UserProcess das System ohne diesen starten und mit der default Variante fahren. Da unsere Beispielimplementation klein ist und wenig Speicher benötigt, nahmen wir den zusätzlichen Flash - und RAM - Verbrauch in Kauf.

8.5 Mögliche Funktionen des UserProcesses

Der UserProcess kann z.B. Timestamps an Pakete anfügen, um dem Client den genauen Zeitpunkt der Messung mitzuteilen. So kann man die Verzögerung der Kommunikation zum Client kompensieren. Da der Stack von verschiedenen Prozessen simultan und ohne Kenntnis voneinander verwendet werden kann, kann der UserProcess von sich aus TCP oder UDP Verbindungen unter Umgehung des Serverprozesses zu einem Client aufmachen: entweder für einen permanenten Datenstrom oder um eine Interrupt - Fähigkeit zu erhalten (Alternative zum Polling des Applets). Allerdings darf er nicht denselben Port verwenden wie der Server (Port 80). Als eine andere Art der “Interrupt - Fähigkeit” ist E-Mail Versand möglich, z.B. wenn Jabba als Schnittstelle zu einem Drucker gebraucht wird und einen Papiermangel selbständig der zuständigen Stelle melden soll. Die von Gerät empfangenen Daten können vorverarbeitet werden, z.B. durch Mittelwertbildung.

8.6 Fazit:

Die Struktur von Jabba mit ServerProcess und UserProcess ist eigentlich schon in unserem ursprünglichen Definitionspapier beschrieben. Bei Mettler gab es anfangs Opposition gegen unsere HTTP Lösung, weil die Einsatzmöglichkeiten eingeschränkt würden. Wir konnten das Konzept jedoch verteidigen. Es hat sich bis jetzt sehr gut bewährt und es zeigte sich, dass alle nötigen Funktionen ohne Einschränkungen implementierbar sind. Zum Beispiel bietet Mettler zu ihren Waagen kleine Drucker an, um Messwerte zu protokollieren. Waage und Drucker werden über einen CAN - Bus verbunden. Mit einer Jabba - Box an den beiden Geräten benötigt man den CAN - Bus nicht mehr und der UserProcess bietet genügend Platz um diese Funktionalität einzubauen. Es sieht im Moment so aus, als ob unser Konzept schlussendlich voll übernommen würde.

9.0 Arbeitsbericht: Waterloo TCP-IP Stack

9.1 Ausgangslage

Uns stand ein Waterloo TCP/IP Stack von Erick Engelke, Version 9606 (also vom Juni 1996) zur Verfügung, welcher von Thomas Hochueli im Rahmen einer Diplomarbeit am Technikum Rapperswil auf den 68000 Prozessor und das OSE RTOS portiert wurde.

9.2 Merkmale

Der Waterloo Stack ist unter der GNU Public License frei auf dem Internet verfügbar. Er war ursprünglich für DOS entwickelt worden, also ein Single Task OS. Entsprechend ist seine Struktur: er besteht lediglich aus einer Bibliothek von Funktionen und einer globalen Datenstruktur pro “Socket”. Es existiert also kein dedizierter TCP Task, alle nötigen Funktionen werden direkt unter der Regie des Applikationstask ausgeführt. Insbesondere muss tcp_tick() regelmässig aufgerufen werden, um das generelle Sockethandling zu vollziehen (Retransmissions und Timeouts, Abarbeitung von ankommenden IP - Paketen). Z.B. nimmt die Funktion sock_send() einen Data-Pointer und eine Länge entgegen, füllt die Daten in verschiedene IP Pakete und ruft die Senderoutine des Ethernet Treibers auf. Erst dann “returnt” sock_send(). Wenn die IP Pakete nicht sofort gesendet werden können, so bleiben sie in der dem Socket zugehörigen globalen Datenstruktur erhalten und werden später von dem regelmässig aufgerufenen tcp_tick() nochmals zu senden versucht.

Die portierte Version 9606 ist schon recht alt. Gemäss den Angaben von Thomas Hochuli funktioniere aber alles ok, bis auf ein paar kleinere Probleme. In der Demonstrationsapplikation zu seiner Arbeit benötigte er allerdings immer nur eine einzige TCP Verbindung aus einem einzigen Task heraus. Zudem sendete der NIC - Treiber für jedes ankommende Paket ein Signal an die Applikation; er musste also über deren Existenz Bescheid wissen, was in unserem Fall nicht erwünscht war. Für uns stellte sich die Frage, ob und mit welchem Aufwand wir diesen TCP/IP Stack für unser Projekt verwenden können und ob wir allenfalls unsere SW Struktur an den Stack hätten anpassen müssen.

9.3 Fragen

- Stabilität: läuft der Stack stabil genug?
- Multitask Umgebung: Um den Programmierkomfort unseres RTOS auszunutzen und eine gewisse Modularität zu erreichen ist es nötig, die Stackfunktionen aus verschiedenen Tasks heraus aufzurufen. Gibt es da Konflikte? Brauchen wir einen dedizierten Task für das TCP Handling?

- Mehrere TCP-Verbindungen: Insbesondere bei HTML Download fallen mehrere Verbindungen gleichzeitig an. Sind wir in der Lage, diese quasi simultan zu behandeln oder müssen wir sie sequentiell abarbeiten? Oder in anderen Worten, können wir mehrere “Sockets” verwalten?
- Geschwindigkeit / Ressourcen: Ist der Stack schnell und klein genug für unsere Anforderungen?

9.4 Demoapplikation Mosiapp.cpp

In dieser Demoapplikation versuchten wir, die obigen Fragen zu beantworten und den noch zu leistenden Aufwand abzuschätzen. Zudem war es eine gute Gelegenheit, um die Programmierumgebung, den Lauterbach Source Level Debugger, die Entwicklungshardware, das OSE RTOS und den Waterloo TCP/IP Stack kennenzulernen. Wir nahmen uns dafür etwa zwei Wochen Zeit, da die Ergebnisse die genauere Struktur unserer Arbeit wesentlich beeinflussen würden.

Wir demonstrierten hier, dass mehrere TCP Verbindungen auch in der portierten Version möglich sind. In einem rudimentären Belastungstest wiesen wir genügende Stabilität und Geschwindigkeit nach. In der Dokumentation, die allerdings recht spärlich war, fanden wir Hinweise, dass der Stack auch in einer Multitaskingumgebung lauffähig sei. Wir definierten daraufhin die entgültige Struktur unseres Projektes:

- Je ein Task für den Server (JabbaSrv) und den UserTask (JabbaUsr), wie in der Projektdefinition vorgeschlagen.
- Wir entschieden uns gegen einen dedizierten TCP Task aus zwei Gründen: erstens entsprach dies am besten der Natur des Stacks, und zweitens ist so unsere Arbeit einfach auf andere Betriebssysteme oder gar auf einen Microcontroller ohne Betriebsystem portierbar. Zudem kann ein weiterer Kommunikationstask zu unserem Projekt hinzugefügt werden, ohne dass der Rest irgend etwas davon wissen müsste (z.B. kann so ein Telnet Server selbständig einen eigenen Socket auf den Port 23 setzen, ohne dass dies den HTTP - Server interessiert).
- Fünf statisch allozierte “Sockets” für den Server, um fünf TCP Verbindungen gleichzeitig offen zu halten. Dieser Wert ist jedoch an einer Konstante einstellbar, für spätere Optimierungen.
- Bei einem ankommenden Ethernet Paket kopiert ein Treiberinterrupttask die Daten nur in einen Puffer und sendet ein Signal an einen separaten Task (“NIC Task”). Dieser führt dann in alle weiteren notwendigen Operationen aus, sprich er ruft tcp_tick() auf. Dieses Vorgehen dient lediglich dazu, den Interrupttask zu entlasten, da die Funktion tcp_tick() zu umfangreich ist.

9.5 Aufgetretene Probleme

In der Folge traten verschiedene unerwartete Probleme auf, welche uns zwangen, mehr Zeit in den TCP Stack zu investieren als angenommen.

9.5.1 Stockende Datenübertragung

Das Stocken war visuell gut bemerkbar bei grösseren Bildern. Ethernetpakete wurden teilweise zweimal geschickt, es gab falsche SEQ und ACK Nummern, falsche Checksummen und undefinierbaren Netzwerkverkehr. Da eine Neusynchronisierung von Sender und Empfänger seine Zeit braucht und z.T. die Verbindungen auch zurückgesetzt wurden, stockte die Datenübertragung.

9.5.2 Unsauberer Verbindungsabbruch

Wenn die Verbindung vom Jabba Server her ordnungsgemäss abgebrochen wurde, so wurde sie in etwas 30% der Fälle vor dem Schliessen der zweiten Simplexverbindung (Client -> Server) vom Server zurückgesetzt (RST).

9.5.3 "FIN" Überschwemmung

Wenn der Client von sich aus die Verbindung mit einem FIN Paket abbrach, entstand des öfteren ein riesen Rattenschwanz von FIN Paketen zwischen Jabba Server und Client hin und her.

9.5.4 Unnötig ACK Pakete

Unnötige ACK Pakete vom Jabba Server. Es wurden auch ganz normale ACK Pakete des Client vom Server bestätigt.

9.5.5 Endless Loop bei hoher Belastung

Wenn man den Jabba Server einer hohen Belastung aussetzte, z.B. von zwei Clients aus mit einem Browser kontinuierlich HTML Seiten neu lud (shift reload), konnte es passieren, dass der Server blockierte. Auf der seriellen Schnittstelle bekam man nun konstant die Meldung: "smc_write timeout". Diese Meldung stammte vom auf OSE portierten Linuxtreiber für den SMC Ethernetchip.

9.6 Lösungen / Änderungen

9.6.1 Stockende Datenübertragung

Nach längerem studieren des Netzwerkverkehrs mit dem Shomiti Surveyor fand ich das Problem: wir hatten zur Demonstration eine zweite Jabba Box an unserem Inselnetz. Da zu dieser Zeit die Mac und IP Adressen noch im Quellcode gesetzt wurden, waren also zwei Instanzen mit denselben Adressen am Netz. (UUPS!) Klar, dass undefinierbare Resultate die Folge waren. Ich musste erfahren, dass Fehler offenbar nicht immer genau sind, wo man sie sucht! Nach einer Neukonfiguration, resp. nach Abhängen der Demobox funktionierte alles besser und schneller, wir hatten keine falschen Checksummen mehr und der Netzverkehr sah viel geordneter aus.Allerdings stimmten sporadisch die ACK und SEQ Nummern nicht, und auch das Laden eines Bildes stockte noch immer. Nach langwieriger Fehlersuche mit Debug Meldungen von überallher aus dem Stack kam ich zum Schluss, dass es sich um ein Synchronisationsproblem zwischen dem Server Task und dem Interruptentlastungstask, dem "NIC Task", handeln musste. Ich stellte mir das so vor: grössere Datenmengen wurden z.B. vom Server am Stück mit der Funktion sock_send() dem "Socket" übergeben. Dieser teilte sie in 2k grosse Buffer - Blöcke und übergab sie der Funktion tcp_send(). Nachdem nun die erste Hälfte im ersten 1.5k

Ethernetpaket versendet worden war, konnte es vorkommen, dass dessen Bestätigung vom Client eintraf, bevor die zweite Hälfte fertig verschickt wurde. Nun unterbrach der Interrupttask tcp_send(), kopierte das Bestätigunspaket in einen Puffer und sendete ein Signal an den NIC Task (Entlastung des Interrupttask). Offenbar kam nun dieser zuerst an die Reihe. Er führte ein tcp_tick() aus, welches die ACK Nummer unseres TCP "Sockets" entsprechend modifizierte. Nach dessen Beendigung konnte unser ursprüngliches tcp_send() mit dem Verschicken der zweiten Hälfte fortfahren. Es fand nun eine modifizierte ACK Nummer vor, überschrieb diese jedoch mit seiner eigenen. Hier hatten wir den Konflikt und die falsche ACK Nummer. Als erstes verkleinerte ich die 2kB grossen Buffer - Blöcke auf 1kB. Sie passten nun ohne Fragmentierung in ein Ethernetpaket. Und schon war das ursprüngliche Problem behoben. Allerdings war die Übertragungsrate in den Keller gefallen und zudem hatte ich auch keine Garantie, dass die 1kB Pakete nicht doch fragmentiert wurden. Ich sperrte also den Zugang zu den kritischen Bereichen von tcp_send() und tcp_tick(), wo die SEQ und ACK Nummern gelesen oder geschrieben wurden, mit einer Semaphore. Nach zurücksetzen der Buffer - Blöcke auf 2k hatten wir eine konstant schnelle Datenübertragung. Nach den doppelt vorhandenen IP und MAC Adressen war ich eigentlich ganz froh, einen "richtigen" Fehler gefunden zu haben. Ich liess mich zu schnell von der etwas unklaren Aussage der Waterloo Dokumentation überzeugen, der Stack funktioniere in einer Multitaskingumgebung. Deshalb suchte ich den Fehler anfangs nicht dort, sondern am Zustandsautomaten des Stacks. Das Setzen der Semaphore erwies sich als heimtückisch. Ich wollte die Daten nicht zu lange blockieren, als ich aber von verschiedenen Orten her die Semaphore immer nur kurzzeitig anwendete, blieb der gesamte Server sporadisch stehen. Unter bestimmten Umständen beanspruchte ich den Speicherbereich für mich und, bevor ich ihn wieder freigab, tat ich das selbe nochmals. Und so wartete ich dort ewig...

9.6.2 Unsauberer Verbindungsabbruch

Dieses Problem ist nicht im eigentlichen Sinn ein Problem, dennoch ist es unschön und deutet auf Fehler im Stack hin. Ich ging deshalb der Sache nach. Der Fehler lag in einem viel zu kleinen Timeout für Verbindungsabbrüche. Wenn der Client nicht sofort das FIN unseres Servers reagierte und unmittelbar danach sein eigenes FIN schickte, erhielt der

```
24.572244   64 TCP DP=80 SP=1082 SYN SEQ=52402 ACK=0 LEN=0 WIN=8192 OPT
24.574571   64 TCP DP=1082 SP=80 SYN SEQ=47912 ACK=52403 LEN=0 WIN=2048 OPT
24.574773   64 TCP DP=80 SP=1082     SEQ=52403 ACK=47913 LEN=0 WIN=8400
24.576324  404 HTTP Message Type: Full Request
24.586995   64 TCP DP=1082 SP=80     SEQ=47913 ACK=52749 LEN=0 WIN=2048
24.599680   75 HTTP Message Type: Full Response
24.617494 1458 HTTP Message Type: Simple Response
24.617753   64 TCP DP=80 SP=1082     SEQ=52749 ACK=49330 LEN=0 WIN=8400
24.623385  374 HTTP Message Type: Simple Response
24.625415  374 HTTP Message Type: Simple Response
24.626532   64 TCP DP=1082 SP=80     SEQ=51063 ACK=52749 LEN=0 WIN=2048 PSH
24.785157   64 TCP DP=1082 SP=80     SEQ=51063 ACK=52749 LEN=0 WIN=2048 PSH
24.819172   64 TCP DP=80 SP=1082     SEQ=52749 ACK=49646 LEN=0 WIN=8084
24.821559   64 TCP DP=1082 SP=80     SEQ=51063 ACK=52749 LEN=0 WIN=2048
24.822880   64 TCP DP=1082 SP=80 FIN SEQ=51063 ACK=52749 LEN=0 WIN=2048
24.823022   64 TCP DP=80 SP=1082     SEQ=52749 ACK=49646 LEN=0 WIN=8084
25.871456   64 TCP DP=1082 SP=80     SEQ=49646 ACK=52749 LEN=0 WIN=2048

Im letzten Paket ist das RST Flag gesetzt, um die Verbindung zurueckzusetzen.
```

Jabba Server eine Zeitüberschreitung und er setzte die Verbindung mit einem RST Paket

zurück. Nach Erhöhung des Timeoutwertes war der Fehler behoben. Ein Source Level Debugger ist zwar eine tolle Sache, für zeitkritische Fälle ist er jedoch keine grosse Hilfe, da die Systemzeit des Clients ja nicht angehalten werden kann.

9.6.3 "FIN" Überschwemmung

Bei der Analyse des Netzwerkverkehrs bemerkte ich falsche ACK Nummern des Servers. Wenn der Jabba Server und der Client zur selben Zeit die Verbindung schliessen wollten und sich so die ersten FIN Pakete kreuzten, verschob sich die ACK Nummer des Jabba Servers um eins. Da die Synchronisation nicht mehr stimmte, versuchten beide Seiten immer wieder erfolglos, die Verbindung zu schliessen, bis endlich eine Zeitüberschreitung eintrat und die Verbindung zurückgesetzt wurde (RST - Flag). Während dieser Zeit wurden mindestens 30 FIN Pakete hin und hergesendet. Dies belastete nicht nur das Netz und unseren Stack, sondern blockierte auch den entsprechenden "Socket" viel zu lange. Wir vermuteten hier einen Fehler in der Zustandsmaschiene des Stacks beim Schliessen der Verbindung. Durch Debug Meldungen über den aktuellen Zustand konn-

```
44.918754   64 TCP  DP=80 SP=3260 SYN SEQ=68390 ACK=0 LEN=0 WIN=8192 OPT
44.921223   64 TCP  DP=3260 SP=80 SYN SEQ=47244 ACK=68391 LEN=0 WIN=2048 OPT
44.921407   64 TCP  DP=80 SP=3260     SEQ=68391 ACK=47245 LEN=0 WIN=8400
44.923573  326 HTTP Message Type: Full Request
44.945803   64 TCP  DP=3260 SP=80     SEQ=47245 ACK=68659 LEN=0 WIN=2048
45.443676   75 HTTP Message Type: Full Response
45.491674  141 HTTP Message Type: Simple Response
45.491878   64 TCP  DP=80 SP=3260     SEQ=68659 ACK=47345 LEN=0 WIN=8300
45.494372   64 TCP  DP=80 SP=3260 FIN SEQ=68659 ACK=47345 LEN=0 WIN=8300
45.494497   64 TCP  DP=3260 SP=80     SEQ=47345 ACK=68659 LEN=0 WIN=2048
45.495816   64 TCP  DP=3260 SP=80 FIN SEQ=47345 ACK=68659 LEN=0 WIN=2048
45.495987   64 TCP  DP=80 SP=3260     SEQ=68660 ACK=47346 LEN=0 WIN=8300
45.567507   64 TCP  DP=3260 SP=80 FIN SEQ=47345 ACK=68659 LEN=0 WIN=2048
45.567650   64 TCP  DP=80 SP=3260     SEQ=68660 ACK=47346 LEN=0 WIN=8300
45.643375   64 TCP  DP=3260 SP=80 FIN SEQ=47345 ACK=68659 LEN=0 WIN=2048

FIN Schwemme: es folgen noch viele Synchronisationsversuche bis zum RST.
```

ten wir den Fehler eingrenzen. Es schien, dass die SEQ Nummer beim Übergang vom Zustand FIN _WAIT_1 nach CLOSING nicht um eins erhöht wurde, sondern nur die ACK Nummer. Dieser Übergang ist ein Spezialfall, der nur bei simultanem Senden des ersten FIN Paketes von beiden Stationen eintreten kann (FIN-FIN ACK-ACK). Es müssen daher beide Zähler erhöht werden um dem Gegenüber ein reguläres FIN-ACK FIN-ACK Spielchen vorzugaukeln. Es war also nicht ein Fehler in der State Machine selbst, sie war immer im richtigen Zustand, sondern ein Fehler in der Nummerierungslogik der Pakete. Nach der Korrektur war das Laden von komplexeren HTML - Seiten spürbar schneller, und es wurde eine grössere maximale Pollingfrequenz erreicht.

9.6.4 Unnötig ACK Pakete

Die unnötigen ACK Pakete beeinflussten weder die Stabilität noch die Effizienz unserer Versuchsanordung wesentlich. Dennoch ist es unschön und es würde z.B. bei einer Verbindung unter zwei Jabba - Boxen zu einer ACK Paketeüberschwemmung führen, da jedes ACK nun ja von beiden Seiten bestätigt würde. Bevor ich hier erneut in die Niederungen des TCP Stacks tauchte, suchten ich www.deja.com nach Meldungen zu diesem Thema ab. Tatsächlich war das Problem bekannt und in neueren Versionen des Waterloo

Stacks schon behoben (z.B. WATTCP_99_11). Die meisten dieser Änderungen machte ein gewisser S. Lawson, wie an jeder relevanten Stelle vermerkt ist. Ich konnte nun aber nicht einfach auf eine neuere Version wechseln, da sie für Intel i386 geschrieben wurde und nicht in unser RTOS passte. Deshalb übernahm ich die Änderungen von S. Lawson manuell in unseren eigenen Stack, allerdings ohne seine Modifikationen an der Socketstruktur, da dies wohl einen Rattenschwanz nach sich gezogen hätte. Nun lief alles fehlerfrei. Alle Änderungen am Stack wurden in der Datei wattcp.cpp vorgenommen und sind mit dem Kommentar “tm” (ich) und “tms” (S.Lawson) vermerkt. In der Version 9911 ist auch der obige Fehler mit der FIN Überschwemmung behoben. Wenn ich mich früher für Hinweise aus Newsgroups bemüht hätte, hätte ich viel Zeit sparen können beim Suchen des letzteren Fehlers.

9.6.5 Endless Loop bei hoher Belastung

Dies ist nicht ein Problem des Stacks, sondern des NIC Treibers (Network Interface Controller). Unter Umständen wird der Fall, dass der NIC keinen freien Speicher mehr hat, vom Treiber nicht richtig abgedeckt. Oder der NIC hat wirklich kein Memory mehr, aus was für einem Grund auch immer. Ich bin diesem Fehler nur kurz nachgegangen, da er nur selten, unter extremen Bedingungen eintrat und die Zeit langsam knapp wurde. Er ist aber sicher auf eine allfällige TODO Liste zu setzen.

9.7 Fazit

Schlussendlich hatten wir einen lauffähigen Stack. Im Testbetrieb fand ich keine Fehler mehr. Die Geschwindigkeit ist besser als erwartet und genügt unseren Anforderungen. Sehr grosse Datenmengen können wegen dem begrenzten Speicheplatz auf der Jabba - Box gar nicht übertragen werden. Wünschenswert wäre eine schnellere Reaktionszeit, die jedoch durch die Verzögerungen eines TCP/IP Netzwerkes gegeben wird. Hier fordert unsere Implementation mit HTTP ihren Tribut. Zur Abhilfe könnte z.B. ein spezieller UserProcess UDP Pakete versenden, somit würde der ganze Verbindungsauf- und abbau entfallen. Da der Stack keine Minimalimplementation ist, bietet er sehr viele Funktionen.

Wenn Mettler unsere Implementation mit dem Waterloo Stack weiterführen will, so stellen sich trotzdem ein paar Probleme. Man hat keine Garantie, dass das Waterloo Projekt weitergeführt wird, und selbst wenn, muss man die eigene 68000 und OSE Variante selbst unterhalten. Irgend wann einmal stellt sich dann auch die Frage nach IPv6. Ob Waterloo einmal IPv6 unterstützen wird ist nicht klar, so oder so wäre es ein grosser Arbeitsaufwand.

Nach den beiden obigen Protokollen von fehlerhaftem Netzwerkverkehr hier noch ein Beispiel des Endresultates. Der Client forderte ein ca. 2k grosses Datenpaket an. Es

```
76.486746   64 TCP  DP=80 SP=1295 SYN SEQ=52934 ACK=0 LEN=0 WIN=8192 OPT
76.489144   64 TCP  DP=1295 SP=80 SYN SEQ=34286 ACK=52935 LEN=0 WIN=2048 OPT
76.489329   64 TCP  DP=80 SP=1295     SEQ=52935 ACK=34287 LEN=0 WIN=8400
76.493523  353 HTTP Message Type: Full Request
76.502740   64 TCP  DP=1295 SP=80     SEQ=34287 ACK=53230 LEN=0 WIN=2048
76.567522   75 HTTP Message Type: Full Response
76.620277 1458 HTTP Message Type: Simple Response
76.620589   64 TCP  DP=80 SP=1295     SEQ=53230 ACK=35704 LEN=0 WIN=8400
76.625963  689 HTTP Message Type: Simple Response
76.628194   64 TCP  DP=1295 SP=80     SEQ=36335 ACK=53230 LEN=0 WIN=2048
76.645960   96 HTTP Message Type: Simple Response
76.646171   64 TCP  DP=80 SP=1295     SEQ=53230 ACK=36373 LEN=0 WIN=7731
76.648805   64 TCP  DP=1295 SP=80 FIN SEQ=36373 ACK=53230 LEN=0 WIN=2048
76.649003   64 TCP  DP=80 SP=1295     SEQ=53230 ACK=36374 LEN=0 WIN=7731
76.711638   64 TCP  DP=80 SP=1295 FIN SEQ=53230 ACK=36374 LEN=0 WIN=7731
76.714026   64 TCP  DP=1295 SP=80     SEQ=36374 ACK=53231 LEN=0 WIN=2048

Korrekter Datenverkehr
```

wird in drei Blöcken gesandt, da es knapp grösser ist als die Fenstergrösse unserer TCP Implementation (identisch mit dem Sendepuffer.)

10.0 Arbeitsbericht: JABBA Konfiguration

10.1 Exploration

Wie konfiguriere ich ein Device, dessen Konfiguration ich nicht kenne? Das Huhn-Ei Problem stellt sich in aller Härte. Es gibt vielfältige Möglichkeiten und Lösungsansätze, dem Problem Herr zu werden. Um die Möglichkeiten etwas zu gruppieren, muss man zwischen automatischer und manueller Konfiguration unterscheiden. Des weiteren sind serielle Schnittstelle und Ethernet getrennt zu beachten.

10.1.1 Automatische Konfiguration

Im Bereich der automatischen Konfiguration von TCP/IP gibt es verschiedene Ansätze. Ich denke an BOOTP, DHCP, TFTP, alles etablierte und standardisierte Verfahren. TFTP und BOOTP würden sogar Möglichkeiten öffnen, eine aktuelle Version der Jabba Firmware vom Netz zu laden.

Eine automatische Konfiguration der seriellen Schnittstelle gibt es meines Wissens nicht. Hier gilt: Die Parameter müssen übereinstimmen, sonst klappt die Kommunikation nicht.

10.1.2 Manuelle Konfiguration

Als manuelle Konfigurationsmöglichkeit von TCP/IP ist uns nur eine Idee gekommen: Abfrage der Konfiguration aller Jabba Devices im LAN via UDP Broadcast. Die neue Konfiguration könnte ebenfalls auf diesem Weg in die Devices gelangen. Um nur ein Device anzusprechen, könnte im UDP Broadcast Packet zusätzlich zur Information die angesprochene MAC-Adresse codiert werden.

10.1.3 Konfiguration

Als dritte Variante bleibt ein voreingestellter Wert. Das Jabba Device könnte mit einer standard TCP/IP Konfiguration ausgerüstet werden, beispielsweise mit nicht gerouteten Privatadressen. Somit könnte nach vorheriger Umstellung eines PC's das Jabba Device direkt über eine gültige IP-Adresse angesprochen werden.

Die Konfiguration der seriellen Schnittstelle ist nur über Ethernet möglich.

10.2 Entscheidungen

Für die Entscheidung des 'richtigen' Konfigurationsverfahren haben mehrere Einflüsse mitgespielt. Wir sind der Meinung, dass Einfachheit einer der wichtigsten Punkte ist. Nicht zuletzt ist auch die vorhandene Zeit ein Faktor, der miteinberechnet werden muss.

Das wohl einfachste Verfahren zur Konfiguration ist die serielle Schnittstelle. Wir definieren in einer Defaultkonfiguration die Parameter dafür. Der Anwender braucht somit nur ein Terminal oder Terminal Emulator auf seinem PC und ein serielles Kabel, Komponenten die meistens bereits vorhanden sind.

Die Umstellung eingestellter Parameter auf dem Terminal oder Terminal Emulator ist sehr einfach zu bewerkstelligen. Zudem haben sich gewisse Standards etabliert. Aus diesem Grund haben wir uns für 19200 Baud, 8 Bits, 1 Stopbit und keine Parität entschieden.

Mit dieser Variante ist es sogar möglich, unser Device mit einem PDA, beispielsweise einem PSION, zu konfigurieren und warten. Wir haben darauf geachtet, dass alle veränderbaren Parameter über die serielle Schnittstelle erreichbar sind.

Die automatische Konfiguration von TCP/IP über BOOTP oder DHCP wäre sehr reizvoll gewesen. Insbesondere, da im von uns verwendeten IP-Stack beide Methoden implementiert sind. Beide Varianten bringen aber etwelche Nachteile. So sind nur die Netzwerk Parameter automatisch konfigurierbar. Es hätte einen zweiten Anlauf gebraucht, um auch allgemeine Parameter zu konfigurieren. Weiter stellen beide Varianten hohe Ansprüche an die Umgebung des 'virtuellen' Kunden. So ist DHCP oft nicht verfügbar oder wird von anderen Personen gemanaged, als jene, die ein Jabba Device installiert. Der schwerwiegendste Nachteil hat uns zum Verzicht auf DHCP und BOOTP bewogen. Der Einsatz von Jabba Devices in einem isolierten TableNet ist sehr wohl denkbar. Im isolierten TableNet gibt es weder Infrastruktur noch geschultes Personal, das diese Aufgabe wahrnehmen könnten.

Als Defaultkonfiguration für TCP/IP haben wir uns für eine fest zugeteilte IP Adresse der Firma Mettler Toledo entschieden. Das Einrichten einer nicht gerouteten Netzwerkadresse ist möglich, jedoch auch mit Nachteilen verbunden. So sind gerade jene privaten Adressbereiche oft in Intranets im Einsatz, und werden dort sehr wohl geroutet. Diesem Problem ist nur mit einem isolierten LAN mit PC und Jabba Devices Herr zu werden. Somit ist eine Defaultkonfiguration eine weitere Möglichkeit, jedoch nur als Zusatz zu verstehen.

Um jederzeit auf eine definierte Konfiguration zu gelangen, haben wir auf unserer Hardware einen Jumper eingebaut. Mittels Überbrücken ist es möglich, eine einkompilierte Konfiguration zu aktivieren. So ist gegeben, das immer von einem definierten Anfangszustand ausgegangen werden kann.

Jabba Devices werden in verteilten Umgebungen eingesetzt. Aus diesem Grund muss es eine Variante geben, um auch remote Jabba Devices warten und konfigurieren zu können. Alle bisher genannten Methoden sind unbrauchbar, um über mehrere Netzwerke hinweg mit Jabba Devices zu interagieren. Die eingebaute System Funktion /sys/serverconfig ist über das HTTP Protokoll zu erreichen. Diese auch Firewall taugliche Kommunikation ermöglicht es, Jabba Devices entfernt zu konfigurieren und zu warten.

Die Variante mit den UDP Broadcast ist sehr verlockend. Diese birgt zwar wiederum den Nachteil, dass sie im isolierten TableNet ohne PC nur bedingt einsatzfähig ist. Dafür ist sie bedingungslos im Firmen-LAN einsetzbar. Leider hat uns die Zeit nicht mehr gereicht, diese Variante zu implementieren und zu testen.

Als letzte Frage stellt sich, wie die Interaktions-Sprache aussehen soll, mit der ein Jabba Device konfiguriert werden kann. Wie beschrieben, haben wir uns auf der Netzwerkebene für HTTP Requests auf eine System Funktion entschieden. Die Parameterübergabe über den QueryString bietet sich hier an. Mit dieser Methode kann die Konfiguration

über ein HTML Formular, die Kommandozeile des Browsers oder ein Java Applet geschehen. Der Browser sendet ja Formulareinträge im QueryString an den Server zurück. Zudem kann der Administrator Konfigurationsänderungen von Hand vornehmen, ohne auf andere Ressourcen zuzugreifen. Als Feedback von Konfigurationsaufrufen kann der vom Server gesendete HTTP StatusCode verwendet werden. Dies ermöglicht wiederum eine manuelle oder automatische Auswertung mittels einer eigenen Applikation.

Die Konfigurationstokens für die Konfiguration via serieller Schnittstelle wie für die server-config Systemfunktion sind die gleichen. Die Ausstattung der Sprache für die Konfiguration via serieller Schnittstelle muss etwas ausgefeilter sein, da hier meistens Menschen ohne vorgeschaltetes GUI interagieren. Mettler Toledo hat hierfür einen internen Standard, MTSICS, geschaffen, den wir übernommen haben. (Siehe [XYZ])

10.3 Ausblick / Zukunft

Die im letzten Abschnitt erwähnte Konfigurationsmethode über UDP Broadcasts gehört zu den 'heissen' Anwärtern für zukünftige Weiterentwicklungen. Weiter sollten folgende Punkte für Weiterentwicklungen beachtet werden.

- Sicherheit: Die Konfiguration geschieht ohne Passwortschutz. Jedermann mit geeignetem Wissen kann ein Jabba Device beliebig umkonfigurieren. Eine Möglichkeit wäre, auch die System Funktionen unter den implementierten Passwortschutz (HTTP Authentifikation) zu stellen.
- Denial Of Service: Auch mit Passwortschutz ist ein Jabba Device risiken einer DOS Attacke ausgesetzt. Alle Funktionen zur Konfiguration und Wartung sollten nur gefilterte Anfragen erhalten. So müssten beispielsweise die IP-Ranges gefiltert werden, woher die angesprochenen Funktionen aufgerufen werden dürften.

11.0 Arbeitsbericht: Java Software

11.1 Übersicht

Die Java-Software besteht aus folgenden Teilen:

- File-Handling
  JSend: Download der HTML-Daten auf Jabba
  JProcessMot: Download des UserProcess auf Jabba
  JFiler: Datengeneration für Jabba
- Applets - JApplet: Test der Funktionen
  JBMT: erstes einfaches Waagenapplet
  JBNuss: Visualisierung der Wägeresultate
- Kommunikation
  JABBAhttp: Klasse zur Kommunikation mit Jabba
- API
  MTBalance: Klasse, die eine Waage abbildet

JABBAhttp ist die immer verwendete Basisklasse zur Kommunikation mit dem Jabba-Tool. Die Klasse MTBalance wird in allen waagen-orientierten Applets verwendet.

11.2 Grundsatzfragen

11.2.1 Generierung der MTSICS-Befehle bei Applets

Ein grosser Streitpunkt war die Verwendung des MTSICS-Befehlssatzes. Die Waagen werden seriell mit diesem Protokoll angesprochen. Nun stellte sich die Frage ob der Userprozess oder das Applet diesen Befehl generieren sollte.

Der wesentliche Vorteil bei der Erzeugung im Applet: der Userprozess wird simpel und universell, er braucht nur die empfangenen Daten auf die andere Seite zu geben.

Der grosse Nachteil: es können keine effektiven Mehrfachzugriffe implementiert werden und einige (für Mehrfachzugriffe sehr ungeeignete) MTSICS-Befehle müssen entfallen. Es sind dies diejenigen welche spontan, also ohne Anfrage, Waagendaten ausgeben.

Wir haben schon in der Aufgabestellung geschrieben, dass wir gerne nur einen kleinen Userprozess als Demo implementieren wollten. Weiter sprach für einen kleinen Userprozess dass wir kein Protokoll erarbeiten mussten zwischen Applet und Jabba. Dies hätte lange Abklärungen mit den Leuten von Mettler erfordert um einigermassen zukunftsfähig zu sein.

Der kleine Userprozess zusammen mit der MTBalance-Klasse ergibt nun eine sehr angenehme Lösung.

Diese Klasse erlaubt nämlich das Tauschen der darunterliegenden Schicht ohne das Wissen des Appletprogrammierers. D.h. auch die Anpassung an einen neuen UserProcess.

11.2.2 Format des File-Systems auf Jabba

Ein echtes Filesystem oder eine vorbearbeitete Datenstruktur mit allen Dateien waren die alternativen für das File-Management.

Um den Aufwand an Software klein zu halten bietet sich die Vorverarbeitung der Dateien auf dem PC an. Allerdings können somit keine einzelnen Dateien mehr auf Jabba bearbeitet werden, es muss immer der gesamte Inhalt gewechselt werden. Da dies nicht lange dauert und die Software viel kleiner wird, haben wir uns für die verwendung eines monolithischen File-Datenblocks entschieden der sowohl File-Tabelle als auch die Files selber enthält.

Um aber nicht immer den Userprozess miteinbinden zu müssen, wird dieser eigenständig behandelt.

11.2.3 Applet API

Nach den Diskussionen um den Userprozess war klar, dass der Appletprogrammierer eine von der Waage unabhängige Schnittstelle brauchte. Das objektorientierte Java bietet hier eine sehr schöne Lösung: Das an Jabba angeschlossene Objekt wird als Klasse nachgebildet. Somit kann ein Waagentausch durch das Austauschen der MTBalance-Klasse in der Software nachvollzogen werden.

Diese Lösung ist nun auch auf Mettler-Seite beliebt geworden.

11.3 Implementationen

11.3.1 JSend

Hier gab es keine Alternativen zum Einlesen von der Platte und Versenden via JABBAhttp. Dabei lernte ich zum ersten Mal die Filehandling-Funktionen von Java kennen.

11.3.2 JProcessMot

Das Motorola S-File wird geparst, die Daten in ein Array geschrieben und dies anschliessend via JABBAhttp auf JABBA geschrieben.

11.3.3 JFiler

Um das C-Programm auf Jabba zu vereinfachen habe ich mich entschieden die Null-Terminierung der Strings ins File zu schreiben. Somit kann einfach ein Pointer auf den Anfang gesetzt werden und dann mit den Standard-Stringfunktionen gearbeitet werden.

Die Adressberechnung der einzelnen Dateien im File-Datenblock setzte einen Platzhalter im Header oder eine nachfolgende Adress-Tabelle voraus. Um das File übersichtlich

zu gestalten habe ich mich für einen Platzhalter fester Länge im Header entschieden. Somit kann das File im Notfall noch von Hand geändert werden.

11.3.4 JApplet

Dieses Applet kann Get & Post von Textdaten testen. Es warf nur ein Problem auf: Sollte der Daten-Upload via Java-Programm oder Web-Formular geschehen ?

Da das Web-Formular noch Zusatzinfos mitschickt, müssten sie noch gefiltert werden. Wir entschieden uns diese Filterung wegzulassen, da der Upload sowieso von einem PC aus erfolgt auf dem Entwickelt wird. Für Massenuploads wäre dann sowieso ein Java-Programm zuständig.

JApplet war mein Testapplet für die JABBAhttp-Klasse. Die Tests führte ich mit Hilfe von CGI-Skipts auf meinem Tardis-Account aus. So konnte ich die Funktion beim Jabba-Test schon garantieren.

11.3.5 JBMT

Dies war unser erstes Applet das auf der MTBalance-Klasse aufbaut. Es sollte als Beispiel für weitere Mettler-Applets dienen. Leider wurde der objektorientierte Ansatz nicht auf Anhieb übernommen.

Die Implementation ergab aber höchstens Diskussionen um das GUI-Layout.

11.3.6 JBNuss

Gedacht für das TIK-Jubiläum im Februar 2000 visualisiert dieses Applet das Einfüllen und “Stibizen” von Nüsschen.

Unterteilt in die Klassen Nuessle und NuesslePot lassen sich hier leicht andere Darstellungsformen implementieren. Die Nuessle Klasse bestimmt die Darstellung, der NuesslePot das Verhalten der “Nüsse”. Ich hatte bei den Redraw/ Update-Routinen für die Grafik einige Probleme. Dabei schien mein Buch etwas veraltet, stimmte doch die Verwendung dieser Funktionen nicht mit meinem Resultat überein.

11.3.7 JABBAhttp

Stellt die HTTP-Get- und HTTP-Post-Funktionen zur Verfügung. Die Implementation für Strings stellte kein Problem dar. Für die Uploads geht das ganze auch noch mit Binärdaten. Es wird von der URL und URLConnection-Class gebrauch gemacht.

11.3.8 MTBalance

Diese Klasse stellte vor allem ein “politisches” Problem dar. Wir wurden in eine Mettler-interne Diskussion (bezüglich MTSICS) gezogen welche in der Hitze des Gefechts zu viel Zeit in Anspruch nahm.

Geglättet haben sich die Wogen eigentlich erst, nachdem wir unsere Variante durchgezogen hatten. Diese besteht aus dieser Klasse mit zugehörigem simplen UserProcess

und besticht vor allem durch schnelle Implementierbarkeit. Sie ermöglicht zugleich die fast vollständige Nutzung des MTSICS-Befehlssatzes. Die Mängel liegen vor allem bei der Messwertdistribution. Eine Überwachung muss die Waage pollen und erhält die Daten nicht spontan von der Waage. Falls dies trotzdem nötig sein sollte, muss der Usertask entsprechend erweitert werden.

12.0 Arbeitsbericht: Flash Programmierung

12.1 Vorgabe

Für das Programmieren des Daten-Flash-Speichers waren neue Routinen nötig.

Es sind dies Block löschen, Identifikation des Chips und Daten schreiben.

12.2 Entscheide

Für die Aufteilung in Daten-, UserProcess- sowie Parameterblöcke werden eigene Routinen verwendet. Somit ruft das Filesystem nur die Daten- oder Process-Schreibroutine auf, diese bestimmt dann wohin die Daten gelegt werden. Ausserdem sollte diese Aufteilung den Datenupload ohne Userprozess-Abschaltung ermöglichen.

Alternativ dazu hätte das Filesystem die Verwaltung übernehmen könne. Der Mehraufwand bringt jedoch keine brauchbaren Vorteile.

Anmerkung: Die Blockstruktur rührt von dem Flashaufbau her: Der Speicherbereich ist in 11 Blöcke unterteilt die je einzeln gelöscht werden können.

12.3 Probleme

Der C-Compiler darf unter keinen Umständen die Speicherzugriffe wegoptimieren. Deswegen müssen alle Variablen mit “volatile” deklariert werden mit denen die Flashzugriffe geschehen.

Weiter mussten die ankommenden Daten Word-Aligned ins Flash geschrieben werden. Dabei wird bei einer ungeraden Anzahl Bytes das letzte Byte provisorisch mit FFhex erweitert und als Word ins Flash programmiert. Entsprechend wird das erste Byte der nächsten Daten die FFhex-Erweiterung überschreiben. Der Wert FFhex rührt vom Flash her: es kann nur Übergänge von 1 nach 0 programmieren.

13.0 Arbeitsbericht: File System

13.1 Ziel

Der Server benötigt natürlich ein kleines File System. Es soll nach Input eines Dateinamens einen Pointer auf den Anfang der Datei sowie deren Länge zurückgeben.

13.2 Entscheide

Um den Filer klein zu halten bietet sich die Möglichkeit der Vorverarbeitung im PC. Ohne dies muss der Filer eine Filetabelle führen, Replacingalgorithmen enthalten und die Daten erst noch im Flash sichern.

Da wir es vor allem auf eine kleine Implementation absahen und der Bedienungskomfort kaum leidet, haben wir uns für den einfachen Code auf Jabba entschieden.Dafür wurde der JFiler geschrieben.

Das “Outfit” des Datenblocks mit den Files sollte auch noch “von Hand” lesbar sein. Daher haben wir uns den Luxus des Return-Zeichens sowie der Nullterminierung geleistet. Insbesondere die Nullterminierung schlägt sich in einem sehr einfachen C-Code nieder.

Das eigentliche Filesystem auf JABBA besteht somit nur noch aus einer Routine welche die File-Table durchsucht und die Adressen herausliesst.

13.3 Probleme

Die Directory-Struktur unter Dos wird durch “\” unterteilt während HTTP den Slash verwendet. Dies hatte ich im ersten Anlauf übersehen.

Um unendliche Namensuche zu unterbinden haben wir uns entschieden max. 300 Files zuzulassen. Nach einigen File-Not-Found-Abstürzen war dies die letzte Änderung am Filesystem.

14.0 Arbeitsbericht: UserProcess Upload

14.1 Ziel

Um das Jabba-Tool perfekt auf jedes Gerät anpassen zu können ist eine Jabba-seitige Schnittstelle nötig. Dazu haben wir den UserProcess vorgesehen. Dieser ermöglicht andauernde Kontrolle der angeschlossenen Maschine.

Das Herunterladen dieses UserProcess vom Ethernet soll möglich sein.

14.2 Entscheide

14.2.1 Prozess versus Funktionen

Analog zur RPCs hätten wir UserFunctions machen können. Dies liesse aber keine andauernde Kontrolle zu. Ausserdem kämen wir damit in den Bereich ähnlicher Projekte.

14.2.2 Einzelner/ Mehrere Prozesse

Um das Betriebssystem nötigenfalls vereinfachen zu können beschränkten wir uns auf einen UserProcess. Damit könnte bei weniger leistungsfähiger Hardware ein statischer Scheduler verwendet werden.

14.2.3 IPC: Alle Daten übergeben

Zwar wird durch die Übergabe aller Connection-Daten vom UserProcess-Programmierer etwas mehr Sorgfalt gefordert, doch dafür kann er auch Verbindungen selber eröffnen.

14.2.4 Upload: Fester Variablenraum

Die Längenbeschränkung von 1kByte für Variablen entsteht durchs Einfachhalten des Betriebssystems. D.h. der Variablenplatz wird mit fester Grösse eingelinkt. Dies ist etwas verschwenderisch, dafür auch ohne OS leicht machbar.

14.2.5 Upload: Ausführung im Flash

Während dem Upload darf der UserProcess nicht laufen, da der Code im zu beschreibenden Flash steht. Das OS stellt keine Routine zur Verfügung um einen Prozess zu löschen. Somit muss der Entwickler sicherstellen dass der Default-UserProcess im anderen Flash aktiv ist ! Dies wäre nur mit grossem Aufwand anders lösbar. Daher entschieden wir uns für die etwas unbequemere, aber durchaus praktikable Methode mit dem Jumper.

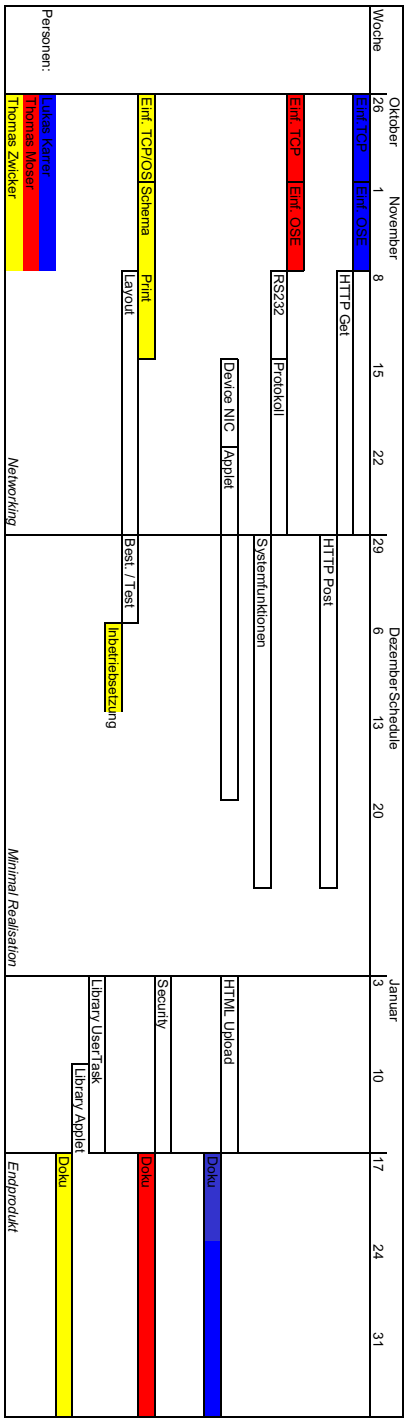
14.3 Probleme

Die Dokumentation des Linkers war nicht eben übersichtlich. Nach einigen E-Mails an den Hersteller liess sich dann aber der UserProcess-Code vom Jabba-Code trennen. Ebenfalls liessen sich die Variablen- und Konstantensektionen desselben separieren. Die Libraries konnten wir bis jetzt aber noch nicht sinnvoll einlinken.

Der eigentliche Upload erfolgt analog zu JSend (siehe Java-Software, JProcessMot).

Das grösste Problem ist das Stoppen des Prozesses. Das OS erlaubt nur dynamischen Start. Daher die Bedingung beim UserProcess-Upload: der DefaultProcess muss aktiv sein !

15.0 Anhang A: Arbeitsplan



16.0 Anhang B: Plakat zum 10 Jahre TIK Jubiläum