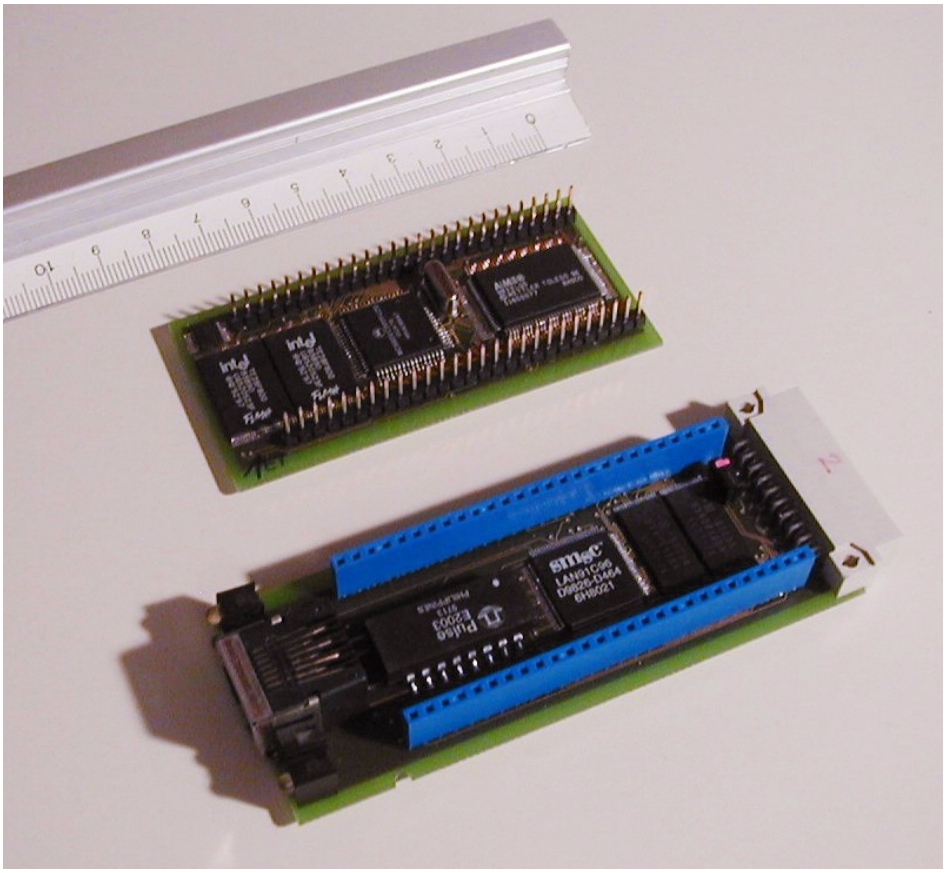
JABBA Embedded Webserver

Lukas Karrer, Thomas Moser, Thomas Zwicker

Technische Dokumentation



Inhaltsverzeichnis

1.0 Einführung

Dieses Dokument beschreibt die Implementation und Funktion von JABBA. Neben dem Hardwareaufbau und der JABBA Software, wird zudem auf die Benützung eingegangen.

Zielgruppe dieser Dokumentation sind sowohl Applet- als auch Firmware-Entwickler, die die Funktionalität von JABBA ausbauen oder verändern möchten.

Neben dieser rein technischen Dokumentation existiert ein zweites Dokument, die das verfolgte Konzept näher erläutert. Es wird empfohlen, insbesondere die Konzeptübersicht zu lesen, um die einzelnen Teile in ihrer Gesamtheit zu sehen.

Sehen Sie dazu: Jabba Entwicklungsbericht, Kapitel 3, Eigener Lösungsansatz.

2.0 Hardware

Die Jabba-Hardware ist eine verkleinerte Version der ProdBox. Sie beinhaltet den M68000 Prozessor, 2*128kByte SRAM, 2*1MByte Flash, den Basco-ASIC, Ethernet Controller und Transceiver. Ein Watchdog sowie je ein 7400 (Inverter) und 7414 (NAND) gehören ebenfalls dazu.

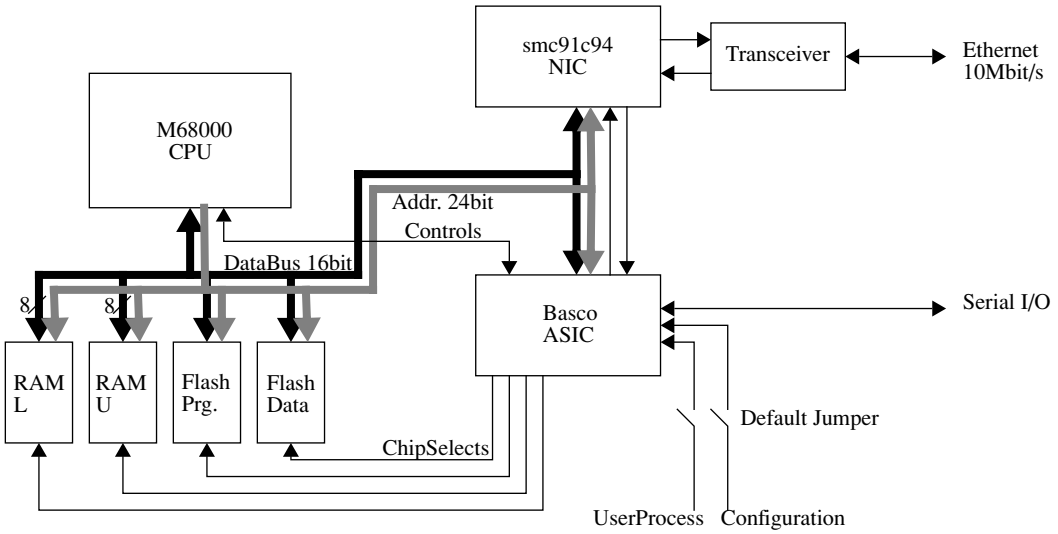


FIGURE 1. Blockschema Hardware

2.1 Prozessor

Der M68000 wird mit 16bit Datenbus betrieben. Dabei werden die Low- & Highbyte-Chipselectsignale verwendet um auf das RAM zuzugreifen, auf die Flash Speicher sind nur 16bit Zugriffe möglich. Das Memory-Timing wird vom Basco gesteuert und kann Waitstates beinhalten. Ebenfalls über den ASIC geht die Interruptkontrolle sowie der DACK. Die Peripherie ist nach Motorola-Art memory-mapped, auch hier generiert der Basco die CS-Signale und somit das Mapping.

Datenblatt: www.motorola.com, MC68EC000

2.2 Address-Mapping

Das Mapping geschieht im Basco. Beim Aufstarten wird das Standard-Flash von Adresse 0 auf 400000H umgeschaltet und das RAM auf Adresse 0 gelegt. Dieser Vorgang ist im Basco-Manual genauer beschrieben. Das Datenflash ist permanent auf 600000H.

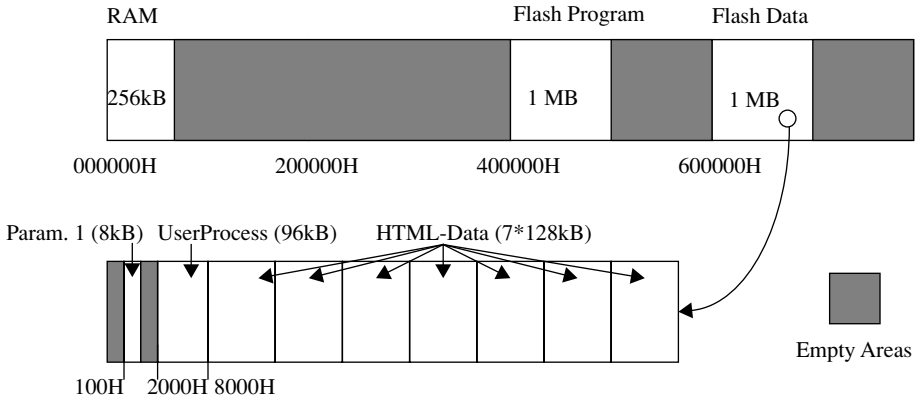


FIGURE 2. Speicheraufteilung

2.3 Ethernet-Controller

Der NIC hat ein M68000er Interface das entsprechend Datenblatt verwendet wird (man beachte die Vertauschung von High- und Low-Byte !). Dazu benötigt er noch einen Transceiver für die Ethernet-Signale. Dies ist ein passives Element zur Signalentkoppelung. Zu Beachten: Der Softwaretreiber muss für diese Hardware angepasst sein. Das Einlesen vom Chip in den Speicher macht bei zu schnellem Zugriff nach Erhalt des Paketes Probleme. Der Optimizer darf die Reihenfolge der Zugriffe nicht vertauschen.

Adressbereich: PCS0 des Basco entspricht 800000Hex

Datenblatt: www.smsc.com, SMC91C94

2.4 Serielle Schnittstelle

Ein Standard-UART ist im Basco implementiert. Die Signalpegel des Ports entsprechen TTL-Pegeln und werden so an den Stecker geführt. Daher der Zusatzprint mit Pegelwandler um Jabba RS232 kompatibel zu machen.

2.5 Flash-Speicher

Diese Flashbausteine sind in 11 Bereiche unterteilt, wovon die tiefste Adresse als Bootblock dienen kann. Weiter enthalten sie 2*8kB-Parameterblöcke, einen 96kB sowie 7 128kB Datenblocks. Zur Programmierung muss:

1. die Programmierquelle eingeschaltet werden
2. der entsprechende Block gelöscht werden (Data = FFH)
3. die Daten geschrieben

4. die Programmierquelle abgeschaltet werden.

Die Codes zur Steuerung der Write-State-Machine sind im Datenblatt ersichtlich.

Baustein: 28F800-5B Datenblatt: www.intel.com

2.5.1 Programm-Flash

Dieser Bootblock ist schreibgeschützt und wird bei der Produktion mit dem Bootloader bespielt. Die anderen Bereiche sind für das Programm, TCP/IP & Server, reserviert. Die Programmierquelle wird mit Basco Port2 Bit 0 eingeschaltet.

Adresse im Betrieb: ab 400000H, Grösse 1MByte, Mapping 2MByte

2.5.2 Daten-Flash

Der Bootblock ist hier nicht schreibgeschützt und kann als Datenspeicher verwendet werden. Die Programmierquelle wird mit Basco Port2 Bit 1 eingeschaltet.

Adresse im Betrieb: ab 600000H, Grösse 1MByte, Mapping 2MByte

2.6 Basco

Dieser ASIC wird in allen LabTec-Waagen eingesetzt. Er enthält das Interfacing für den Prozessor und die Peripherie, UART, AD-Wandler, I2C-Interface und drei I/O-Ports. Der AD- und I2C-Teil wird bei Jabba nicht verwendet. Die I/O-Ports werden bei der Jabba-Hardware analog zur ProdBox wie folgt verwendet:

Port 1:	Input, Bit 0 & 1 für Defaultkonfigurationen an Lötpads
Port 2:	Output, Bit 0 & 1 für Programm-Flash & Daten-Flash Programmierspannungsquellen
Port 3:	Interrupt Inputs, Bit 2 NIC-Interrupt

Für das verwendete RAM benötigt man keine WaitStates, Clockfrequenz = 20MHz. Somit kann der Basco wie für die ProdBox initialisiert werden.

Datenblatt: Projekt Tigris, Dok-Nr. 42034

2.7 Booting, Bootloader

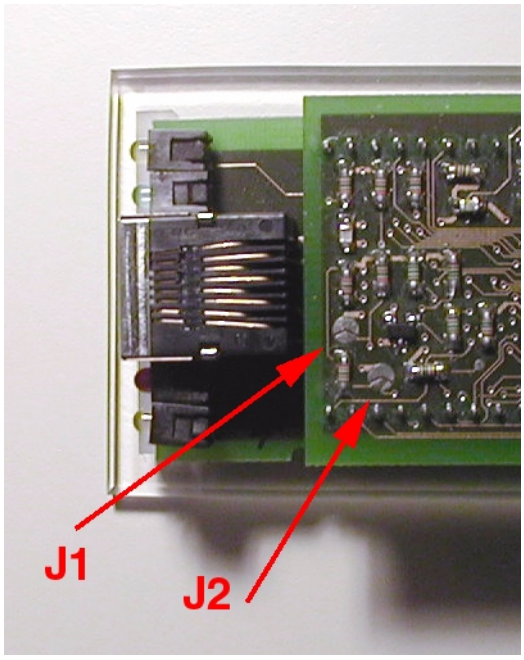
In der Produktion wird dem Programm-Flash ein Bootloader in den Bootblock einprogrammiert. Dieser reagiert in den ersten 500ms auf Bell-Characters die mit 9600baud von der seriellen Schnittstelle kommen. Werden solche Zeichen empfangen, so wird der Bootloader gestartet. Damit lässt sich dann das Programm (Server) auf die Hardware laden. Dies geschieht mit dem Programm Flashloader (ab Version 1.0) von Mettler. Falls keine Daten anliegen kommt das eigentliche Programm zur Ausführung. Eine spezialität des Bootloaders besteht im Umschalten der Flash- & RAM-Adressen. Im

Zusammenspiel mit dem Basco wird nach dem vierten Befehl diese Umschaltung vorgenommen.

2.8 Jumper

Auf der Platine befinden sich 2 Jumper, die über den Basco ausgelesen werden können. Jumper J1 und J2 wird vom JABBA für FailSafe Verhalten verwendet.

FIGURE 3. Jumper auf JABBA Hardware



Kurzgeschlossen bedeuten die Jumper folgendes:

- J1: Setzte Default IP & MAC Adressen bei Reboot
- J2: Starte Default UserProcess bei Reboot

3.0 JABBA Webserver

3.1 Übersicht

Im Jabba Device ist ein vollständiger HTTP Server implementiert. Der Server folgt RFC 1945, die Spezifikation von HTTP Version 1.0. Neben den fest einkompilierten System- und User- Funktionen für den Unterhalt und Betrieb, können ab Jabba auch ganz normale WebSeiten geladen werden.

Der Jabba Webserver unterstützt alle gängigen Dateiformate.

3.2 Webzugriff auf Jabba

Im Jabba Device ist ein einfaches Filesytem implementiert. Das Filesystem wird mittels JFiler Applikation auf einem PC erstellt und via Netzwerk auf das Jabba Device geladen. Das '/' Verzeichnis auf dem Server, das mit http://jabba/ erreichbar ist (DocumentRoot), entspricht dem im JabbaConfig ausgewählten Ordner. Darunter ist die ganze Hierarchie erreichbar, also bspw. http://jabba/Bilder/Image1.jpeg. Zu beachten ist, dass der Zugriff auf die einzelnen Files 'Case Sensitive' ist.

3.2.1 Unterstützte Dateiformate

Prinzipiell sind im Jabba WebServer alle Dateiformate unterstützt. Der WebServer versucht, anhand der Dateiendung den MimeType zu bestimmen. MimeTypes sind nötig, dass der Browser erkennt, wie ein Dokument zu interpretieren ist. (Anzeige der Datei Image.gif als Bild, Dokument.doc in MSWord etc.)

Die gängigsten Datei-Endungen für Text, Bild, Audio und Video sind im Server vorhanden. Wird eine Datei nicht erkannt, so wird der MimeType 'text/html' dem WebBrowser mitgeliefert.

3.2.2 Automatisches Anzeigen einer Index Seite

Wird der Jabba WebServer mit http://jabba/ aufgerufen, so sucht der Webserver nach einer Datei namens 'index.html' im DocumentRoot. Wird diese nicht gefunden, so zeigt der Browser eine Fehlermeldung an.

3.2.3 Eigene HTML Seiten für Fehlermeldungen

Der Webserver kann verschiedene Fehlermeldungen im Browser anzeigen. So erscheint bei einem Aufruf auf eine nicht bekannte Seite eine einfache Fehlermeldung in Textform, die dem Anwender mitteilt, dass die Seite nicht gefunden werden konnte. Der Jabba Webserver ermöglicht es dem Anwender, eigene Fehler-Seiten zu gestalten. Tritt ein Fehler auf, so sucht der Jabba Webserver eine Datei 'FehlerCode.html' im internen Dateisystem. Ist diese vorhanden, so wird die Datei im Browser angezeigt. Ansonsten erscheint die Fehlermeldung in Textform.

Die wichtigsten Fehler-Codes sind 404 -> Seite nicht gefunden und 500 -> Internal Server Error. Es kann jedoch für jeden in RFC 1945 definierten StatusCode eine Seite angelegt werden.

3.2.4 Zugriffschutz einzelner Dateien

Das Jabba Filesystem ermöglicht, einzelne Dateien mit einem Zugriffsschutz zu versehen. Alle Seiten im Jabba Filesystem sind standardmässig mit Attribut R versehen. Dies symbolisiert einen uneingeschränkten Zugriff auf diese Files. Werden Dateien mit Attribut S versehen, so sind diese nur nach Eingabe eines Passworts erreichbar. Der in Jabba.hpp definierte Login und Passwort lautet zweimal JABBA. Login und Passwort können nur vom Programmierer geändert werden. An derselben Stelle können auch die vom Browser angezeigten Credentials geändert werden.

3.3 Jabba Webserver für Programmierer

3.3.1 Datenstrukturen

Jabba HTTP Server verwended einen Pointer auf die Datenstruktur (struct http_sock_type *) s als Basis. Über diesen Pointer sind die zum HTTP Server gehörenden Variablen sowie der mit dem Request assoziierte Netzwerk Socket erreichbar. Jeder eingehende HTTP Request besitzt somit seine eigene Datenstruktur.

Die zum HTTP Server gehörenden Daten, sind einer separaten Struktur (struct HTTP_REQUEST) req eingebettet, die einen Teil der Struktur http_sock_type bildet.

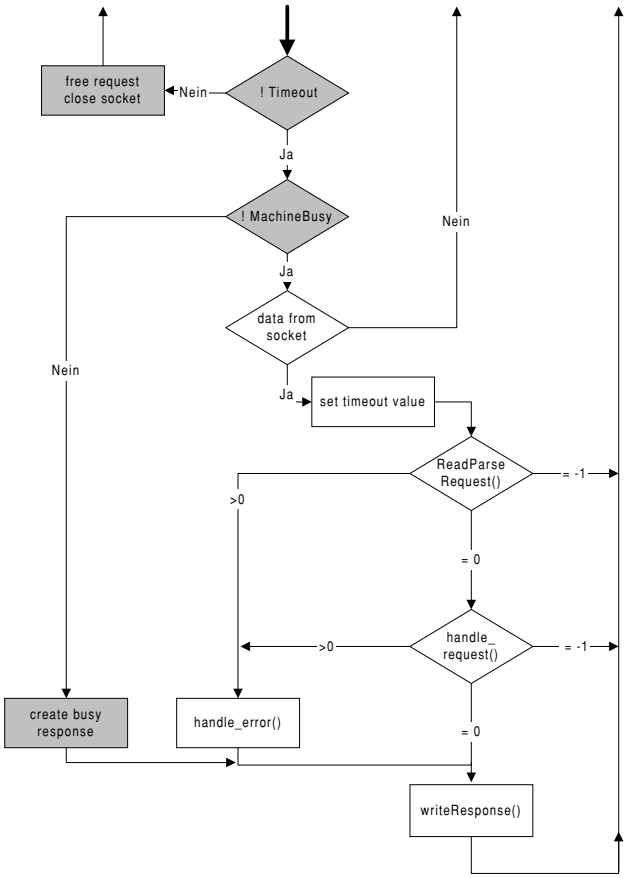
Die Struktur HTTP_REQUEST umfasst alle Daten, die für einen HTTP Anfrage an den Server benötigt werden. Wird eine neue Anfrage an den Server gestellt, so werden die Daten in vorbereitete Variablen in der Struktur geschrieben. Alle nachfolgenden Funktionen, bspw das Handling von Datenupload oder der HTTP Authentifikation operieren auf dieser Struktur. Neben Daten beinhaltet die Struktur zudem mehrere Statusvariablen, bspw. s->req.status, die den Zustand eines HTTP Requests innerhalb des Servers beschrieben.

3.3.2 Rückgabewerte der Implementierten Funktionen

Die meisten Funktionen im Jabba HTTP Server sind vom Typ int. Dies ermöglicht es, dem Aufrufenden den Status mitzuteilen.

Ein ReturnWert von 0 bedeuted, dass die Funktion erfolgreich abgeschlossen hat. Ein positiver ReturnWert signalisiert einen Fehler. Der Server ist so implementiert, dass Fehler mit den standartisierte HTTP Error Codes gekennzeichnet werden. Finded beispielsweise die HandleRequest Funktion die angeforderte Resource nicht, wird sie mit return (404) enden. Der Wert 404 bedeuted nach RFC 1945, "Resource not found”

FIGURE 4. JABBA HTTP Ablaufschema



3.3.3 Grobablauf

Nach der Initialisation der Datenstruktur, pollt der HTTP Server Task den Netzwerk-Socket in regelmässigen Abständen. Sind neue Daten verfügbar, wird die Funktion fHttp_ReadParseRequest aufgerufen, die Anhand des Status des Requests das weitere Vorgehen veranlasst. Ist der HTTP Protokollheader vollständig vom Socket gelesen worden, so übernimmt die Funktion fHttp_HandleRequest die Datenstruktur und arbeitet die Abfrage ab.

Als letzte Instanz schreibt die Funktion fHttp_WriteResponse die in der Struktur gesammelten Daten und schriebt sie zurück auf den Socket. Schliesslich wird der Socket geschlossen und die Kontrolle wieder zurück zum Server Task gegeben.

Dieser Zyklus einer HTTP Anfrage wird in Abb. 4, “JABBA HTTP Ablaufschema,” auf Seite 12 graphisch dargestellt.

Die grau hinterlegten Teile sind zur Zeit noch nicht implementiert, können jedoch jederzeit hinzugefügt werden.

Die aktuelle Version des Jabba HTTP Servers unterstützt noch keinen Timeouts. (Grau hinterlegte Teile von Abb. 4, “JABBA HTTP Ablaufschema,” auf Seite 12) Dies kann bei einem Netzunterbruch inmitten eines FileUploads oder HTTP Anfrage zu Problemen führen.

3.3.4 fHTTP_ReadParseRequest

Abb. 5, “JABBA ReadParseRequest Funktionsablauf,” auf Seite 14 zeigt den Detailierten Aufbau der ReadParseRequest Funktion. fHTTP_ReadParseRequest wird für jedes vom Netzwerk-Socket gelesenen Datenpackets aufgerufen. Anhand des Request-Status s->req.status wird entschieden, ob die gelesenen Daten zum HTTP Protokol Header oder zu Upload Daten gehören. Ist der Header vollständig gelesen und keine weiteren Daten erwartet, so wird die Funktion mit ReturnCode beendet.

Ist die HTTP Anfrage noch nicht vollständig, so wird mit ReturnCode -1 signalisiert, dass noch weitere Daten erwartet werden.

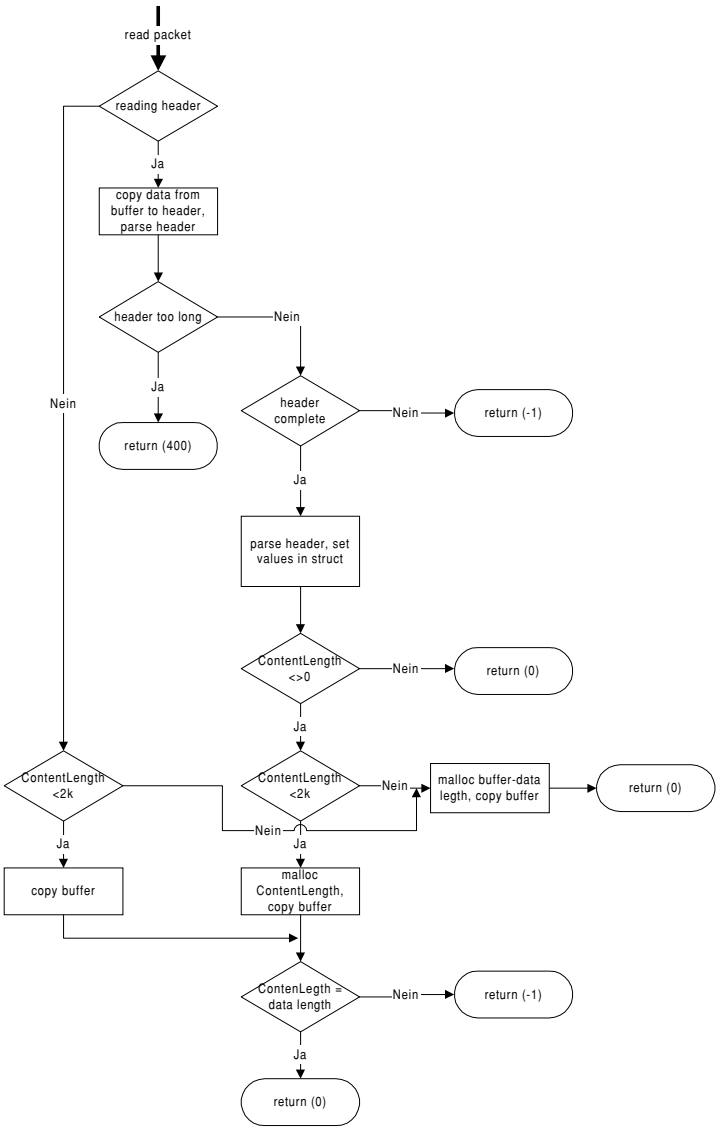
Normalerweise wird fHTTP_ReadParseRequest solange aufgerufen, bis die Anfrage vollständig gelesen wurde. Bei HTTP POST wird die im HTTP Header spezifizierte Grösse Content-Size Memory alloziert und wiederum solange im gleichen Zyklus verbracht, bis alle Daten im Speicher geschrieben sind. Im embedded Bereich ist dieses Vorgehen nur bedingt möglich. Ist die gePOSTete Datenmenge grösser als 2k, so muss speziell verfahren werden. Aus naheliegenden Gründen ist es nicht möglich, 800k Speicher zu allozieren, die der Fileupload benötigen würde.

In Fall von grossen Datenmengen wird nach vollständigem Einlesen des HTTP Headers nur gerade jene Menge Speicher alloziert, die bei dem Polling Durchgang vom Socket gelesen wurde. Die so empfangenen Daten werden an die Funktion fHttp_ReadParseRequest weitergegeben, die selbst wiederum eine Funktion aufruft, die die empfangenen Daten verarbeitet. Die Kontrolle über die Daten sind Sache dieser Funktion. Sie wird mehrfach hintereinander aufgerufen werden, falls der Upload grösser als die definierten 2k ist.

Mit einer Maximum Transfer Unit MTU von 1.5k bei 10BaseT Ethernet sind die gelesenen Datenpackete meist um 1.4k gross.

Die Information über das empfangene Datenpacket wird in der Datenstruktur gespeichert. s->req.iContentLength beinhaltet die totale Grösse des Uploads, s->req.iDataChunkLength die aktuelle Länge.

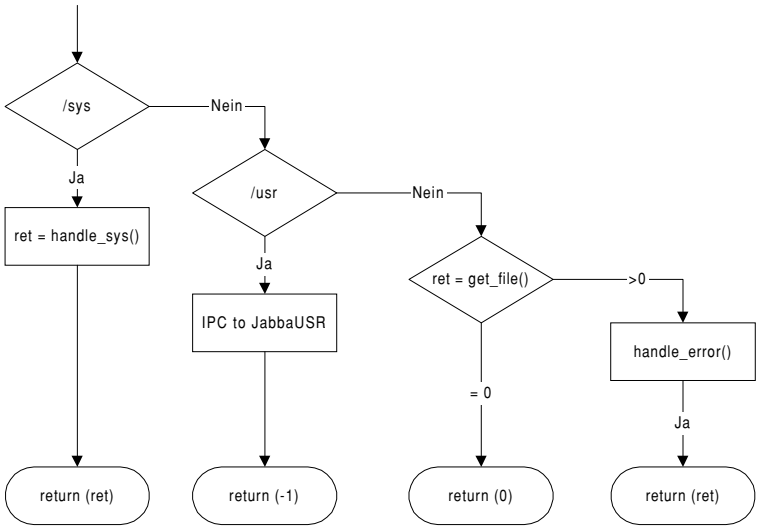
FIGURE 5. JABBA ReadParseRequest Funktionsablauf



3.3.5 fHttp_HandleRequest

fHttp_HandleRequest bestimmt die angeforderte Resource. Enthält die Variable s->req.szPath den String "/sys", so wird der System Function Handler fHttp_HandleSysFunc aufgerufen. Bei einer Anfrage nach "/usr" wird dem UserTask mittels IPC der Pointer auf die aktuelle Datenstruktur struct HTTP_REQUEST übergeben. Trifft keine der zuvorgenannten Bedingungen zu, so wird über fHttp_GetFile aufs Jabba File System zugegriffen. Die ReturnWerte der aufgerufenen Funktionen werden als ReturnWert von fHttp_HandleRequest zurückgegeben.

FIGURE 6. JABBA HandleRequest Funktionsablauf



3.3.6 fHttp_GetFile

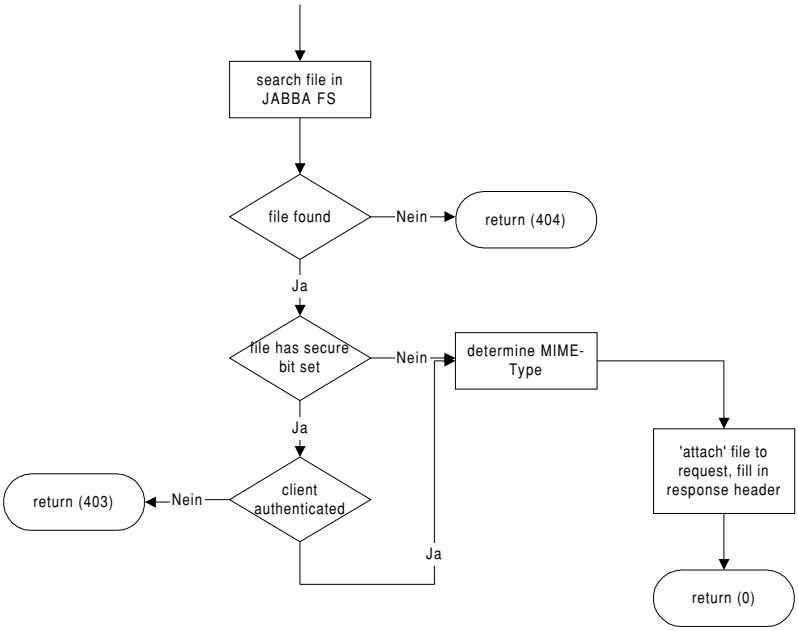
fHttp_GetFile wird aufgerufen, falls der aufgerufene URL weder mit /sys/ oder /usr/ beginnt. Die Funktion sucht im internen Filesystem nach der angegebenen Datei. Ist die Datei vorhanden, so wird anhand der Dateiendung der dazugehörige MIME-Type bestimmt.

Das im JABBA Filesystem vorhandene Secure Bit zeigt an, ob die Datei Passwortgeschützt ist. Dies wird mit dem Buchstaben ‘S’ im signalisiert. Ist dies der Fall, so wird anhand RFC 1945 eine Authentifikation über das HTTP Protokoll ausgeführt.

Der Login und Passwort ist dabei fest in den Programmcode compiliert. Im File Jabba.h definiert die Variable szFilePasswort in der Form “Login:Passwort” die geforderte Eingabe. Abb. 7, “JABBA GetFile Funktionsablauf,” auf Seite 16 verdeutlich den Ablauf in graphischer Form.

Abschliessend werden die Pointer von struct HTTP_REQUEST auf die angeforderte Datei angelegt und die für eine gültige HTTP Antwort nötigen Variablen gesetzt.

FIGURE 7. JABBA GetFile Funktionsablauf



4.0 Jabba Systemfunktionen

4.1 Übersicht

Jabba Systemfunktionen sind feste Bestandteile von Jabba. Sie stehen dem Anwender und Programmierer der Jabba Platform für den Betrieb und Unterhalt zur Verfügung. Aufgerufen werden die Systemfunktionen intern via Funktionsaufruf oder über einen URL. Dieser ist in der Form http://jabba/sys/Funktions-Name.

4.2 Vorhandene System Funktionen.

Folgende Systemfunktionen sind implementiert:

4.2.1 Abruf von Systeminformationen

Der Aufruf von http://jabba/sys/server-info gibt eine HTML Seite zurück, die Informationen und die aktuelle Konfiguration auflistet.

4.2.2 Konfiguration von Jabba

Mittels Aufruf von http://jabba/sys/server-config?TOKEN[=VALUE] kann Jabba über HTTP konfiguriert werden. Die gültigen Token sind in “JABBA Konfiguration” auf Seite 22 beschrieben. Bei einem gültigen Konfigurationsaufruf wird keine Meldung zurückgegeben. Der letzte Teil '=VALUE' ist optional. Wird dieser weggelassen, gibt-Jabba den aktuellen Wert des Tokens zurück.

Achtung: Änderungen eines Werts tritt erst nach einem Neustart von Jabba in Kraft!

4.2.3 Data Upload

Mittels http://jabba/sys/data-upload kann per HTTP Post Anfrage das Jabba eigene Filesystem geladen werden. Das Filesystem ist in “File System” auf Seite 33 beschrieben.

4.2.4 Process Upload

Mittels http://jabba/sys/process-upload kann per HTTP Post Anfrage der Jabba User-Process geladen werden. Siehe dazu Kapitel 10, UserProcess Upload.

Für den Prozess-Upload muss der Jumper J2 Kurzgeschlossen sein !

4.2.5 Memory Dump

Zum Debuggen bietet Jabba eine Memory Dump System Funktion. http://jabba/sys/server-config?[base=BASEADDR][+format={c|x}] Die Ausgabe erfolgt in hexadezimaler (x) oder character byte (c) Schreibweise ab Adresse BASEADDR, die ebenfalls in hex oder int Notation angegeben werden kann.

4.3 Implementation der Systemfunktionen

Jabba System Functions sind in den Webserver eingebetet. Die Kontrolle über den Aufruf der System Functions übernimmt die Funktion fHttp_handle_sysfunc(), die vom Webserver aufgerufen wird, sobald /sys/ im URL steht. Diese Funktion ruft wiederum die einzelnen System Funktionen auf. Der Rückgabewert der Systemfunktionen wird eins zu eins weitergeben.

Als Parameter erhalten Systemfunktionen normalerweise die in Jabba.h definierte Struktur HTTP_REQUEST * req. Darauf darf vollständig zugegriffen werden.

Die Systemfunktionen müssen ein Integer Rückgabewert haben. Ein Return-Wert von 0 bedeutet, dass die Funktion ohne Fehler abgeschlossen hat. Tritt ein Fehler auf, so kann dieser mit positivem Rückgabewert signalisiert werden. Die Zahl des Return-Wertes wird als HTTP Fehler Code interpretiert, sprich 500 = Internal Server Error etc.

Dem aufrufenden Browser kann mittels dem Pointer req->psContent Daten übergeben werden. req->iContentLength beinhaltet hierfür die Länge der Daten. Der standard HTTP Content-Type text/html kann zudem auf die zu übergebenden Daten angepasst werden. Dies geschieht mittels req->szContentType.

Der aufrufende Browser kann via der POST Methode oder als QueryString Parameter und / oder Daten übergeben. Im ersten Fall zeigt req->psContent auf req->iContent-Length Daten. Im zweiten Fall sind Daten und Parameter über req->szQueryString in URL-Decodierter Form erreichbar.

4.3.1 Skelett einer System Funktion

```
int fSys_MyFunction(  HTTP_REQUEST * req ) {
  // auslesen der Parameter
  if (*req->szQueryString == 0)
    return(501); // Wir wollen einen Parameter -> Fehler 501

  sscanf(req->szQueryString,
  "%[ABCDEFGHIJKLMNOPQRSTUVWXYZ]=%[^&]",szToken, szValue);
  {
    FUNKTION
  }

  if (Fehler)
    return(504);// Internal Server Error
  if (OK && Kein Rückgabert)
    return(203);// No Content
  else
  {
    req->psContent = "Mein Text";
    req->iContentLength = strlen ("Mein Text");
    req->iStatusCode=200;
    req->szContentType = "text/plain";
    return(0);// kein Fehler
  }
}
```

5.0 UserProcess

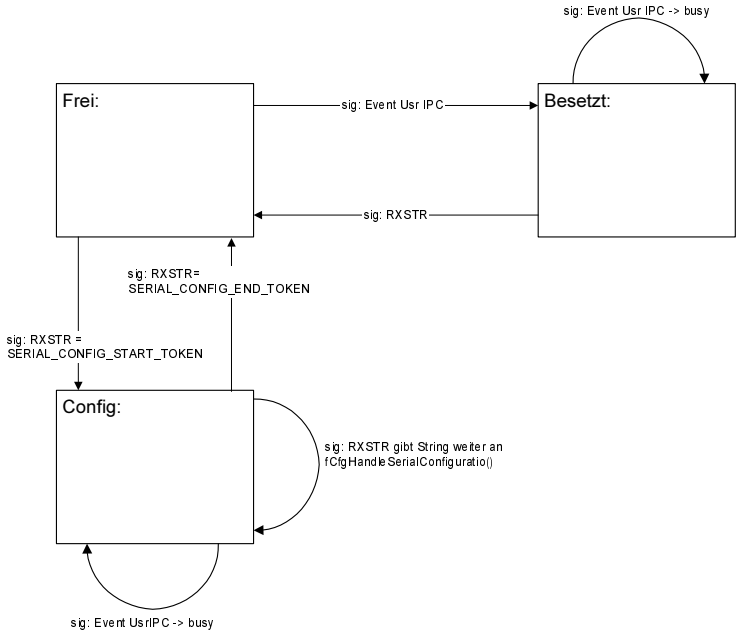
5.1 Zweck des UserProcesses

Die Idee des UserProcesses ist es, dem Programmierer ein hardwarenahes Programmieren ohne genaue System- und Netzwerkkenntnisse zu ermöglichen. Jabba soll auf möglichst hoher Ebene programmiert werden. Ein Grossteil der Funktionen wird deshalb als Applets in Java realisiert. Damit wird die Flexibilität hochgehalten, auf Kosten der Geschwindigkeit und Echtzeitfähigkeit. Wenn nun bestimmte Geschwindigkeits- oder Echtzeitkriterien gefordert werden, so können sie über den UserProcess erfüllt werden. Deshalb stellen wir hier eine zweite, ergänzende Variante der Programmierung zur Verfügung.

5.2 Struktur des UserProcesses

Die Struktur von Jabba sieht zwei eigene Prozesse vor um die Kommunikationsschnittstellen Ethernet und Uart zu bedienen: JabbaSrv und JabbaUsr (=UserProcess). Sie werden über IPC (Inter Prozess Communication) Mechanismen verbunden. Um unsere Jabba Box an eine andere Maschine anzupassen, sind Änderungen allenfalls im UserProcess vorzunehmen. Der Rest von Jabba mit dem Server und allen Konfigurationsmechanismen bleiben somit unverändert.

FIGURE 8. Statemachine des UserProcess



5.3 Mögliche Funktionen des UserProcesses

Der UserProcess kann z.B. Timestamps an Pakete anfügen, um dem Client den genauen Zeitpunkt der Messung mitzuteilen. 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 nicht derselbe Port verwendet werden 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.

5.4 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 fCfg_HandleSerialConfiguration() beispielhaft zeigen. Er ist an die Waagenschnittstelle vom Mettler Toledo (MT-SICS) angepasst, aber dennoch sehr allgemein verwendbar.

5.5 Aufbau unseres Beispiel - UserProcess (DefaultUserProcess)

Unser UserProcess ist im Prinzip ein einfacher Zustandsautomat mit drei Zuständen: Frei, Besetzt und Config. Da die Waage nicht verschiedene Anfragen auf einmal verarbeiten kann, wird deren Zugang gesperrt, während auf eine Antwort gewartet wird. Eine Zeitüberschreitung stellt sicher, dass der UserProcess bei einem Fehler nicht ewig weiterwartet. Der MT-SICS Befehlssatz der Waage ist so aufgebaut, dass auf alle Befehle eine Bestätigung zurückkommt. Im allgemeinen haben wir also immer schnell eine Antwort, ausser beim Tarieren oder wenn ein stabiler Wägewert verlagt wird ("S" anstatt "SI"). Dort wartet die Waage, bis sich alles eingependelt hat. Eine Hand auf dem Tisch kann allerdings schon genügen, um das Einschwingen zu verzögern. Die Zeitüberschreitung von zehn Sekunden ist deshalb ein Kompromiss, der unter normalen Umständen ein Tarieren ermöglichen soll, uns aber beim Entwickeln oder Demonstrieren dennoch nicht zu lange warten lässt.

<Flow chart UserTaskStateMachine>

Wenn der Jabba Server ein HTTP Request in der Form "http://<jabba>/usr/..." bekommt, so füllt er die req Struktur mit den notwendigen Daten und schickt die gesamte http_sock_type - Struktur via EVENT Signal an den UserProcess. In unserem Beispiel benutzen wir die URL nach dem "/usr/" als Informationsträger. Dies, weil man so aus einem Browser oder einen Telnet ohne Sonderzeichen manuell einen Request sehr einfach absetzen kann. Man könnte allerdings auch den Query String gebrauchen, er ist in der req Struktur gespeichert. Der UserProcess bekommt nun also ein EVENT Signal mit der Eventnummer USER_HTTP_IPC. Als erstes schneidet er das "/usr/" ab, und übrig bleibt ein String, der nur mehr den rohen Waagenbefehl beinhaltet. Diesen

schicken wir an die UART, wechseln in den Zustand “Besetzt” und warten. Alle anderen Signale werden jetzt zwar empfangen, um die Signalqueue leer zu halten und damit einen Überlauf zu vermeiden. Sie werden aber einfach ignoriert oder mit einer Fehermeldung “resource busy” quittiert. Wenn vor der Zeitüberschreitung Daten von der Waage zurückkommen, nehmen wir an, es handle sich um unsere Antwort und hängen sie in die req Struktur ein. Via ein Signal erkennt der Server dann, dass die Daten für die HTTP Reply angekommen sind, schickt die Reply und schliesst den Socket. Und wir wechseln wieder in den Zustand Frei.

Im Zustand Frei werden alle Daten von der UART ignoriert, ausser wenn sie den SERIAL_CONFIG_START_TOKEN enthalten. Falls dieser vorhanden ist, wechselt der Zustandsautomat in den Config Mode. Dort werden alle erhaltenen UART Daten an fCfg_HandleSerialConfiguration() weitergereicht. Dieser Parser ruft selbständig die entsprechenden Funktionen auf (übrigens dieselben Funktionen, welche auch bei der Konfiguration via HTTP benutzt werden.) Man kann nun mittels den dokumentierten Befehlen über ein Terminal sämtliche Einstellungen vornehmen (IP-Adressen, MAC-Adressen, Baudrate der Uart, Hostname etc.) Mit dieser Lösung stellen wir sicher, dass eine Änderung oder Ergänzung der Konfigurationsmöglichkeiten von Jabba keinen Einfluss haben auf den UserProcess. Zudem überlassen wir es dem UserProcess, die eigene Konfiguration unabhängig davon selbst vorzunehmen, er kann aber auch bequem die Cfg - Funktionen aufrufen, wenn er den Parser nicht brauchen will (siehe JabbaCfg.hpp). Der Programmierer muss sich also nur um den eigenen Prozess kümmern, vom Rest von Jabba braucht er nicht viel zu wissen. Dies war unser Ziel.

5.6 Referenzen zum UserProcess:

- JabbaCfg.hpp beinhaltet die Konfigurationsroutinen.
- JabbaSys.hpp beinhaltet Systemfunktionen wie z.B. server-info und E-mailversand.
- JabbaUsr.hpp/cpp Beispielimplementation eines generellen UserProcesses.
- Dokumentation WattcpDokumentation des TCP/IP Stacks, falls eigene Verbindungen aufgemacht werden sollen.
- OSE RTOS DokuReferenz zum Betriebsystem. Z.B. für Signals und andere Funktionen.

6.0 JABBA Konfiguration

6.1 Übersicht

JABBA kann über die serielle Schnittstelle sowie über HTTP Aufrufe konfiguriert werden. Die Konfiguration wird im nicht-flüchtigen Speicher abgelegt, sodass auch nach einem Reboot die Konfiguration erhalten bleibt. Ist eine unbekannte Konfiguration aktiviert, so ist es mit dem FailSafe Jumper möglich, die Werkseitig einkompilierte Konfiguration zu aktivieren. Siehe Abb. 3, “Jumper auf JABBA Hardware,” auf Seite 9 für die Anordnung der Jumper.

Achtung: Konfigurationsänderungen werden erst NACH einem Neustart aktiv. Ein Neustart wird entweder per Befehl oder PowerCycle (= ausstecken des Adapterkabels) ausgeführt.

6.2 Konfigurationsvariablen und deren Defaultwerte

In folgenden Kapitel sind die Konfigurationsvariablen und die dazugehörigen Defaulteinstellungen beschrieben. Die Ist der gültige Wertebereich beschränkt, so sind die Auswahlmöglichkeiten in geschweiften Klammern aufgeführt.

6.2.1 Parameter für die Netzwerk Schnittstelle

HOSTNAME jabba
DOMAINmtg.mtnet
IPADDR172.24.113.10
NETMASK255.255.255.0
GATEWAY172.24.112.1
NAMESERVER172.24.112.20
MACADDR12:34:56:78:90:10
SMTPmailhost.mtg.mtnet.

Anmerkung: Fully Qualified Domain Names FQDN müssen mit einem zusätzlichen '.' am Ende gekennzeichnet werden.

6.2.2 Parameter für die serielle Schnittstelle

BAUDRATEBd192 {Bd24|Bd96|Bd192|Bd576}
PARITYParityNo {ParityNo|ParityEven|ParityOdd}
BITBit8 {Bit7|Bit8}
STOPStop1 {Stop1|Stop2}

6.2.3 Zusätzliche Befehle

INFO : Output der aktuellen Konfiguration und Systemvariablen. (Analog zur Systemfunktion http://jabba/sys/server-info)

RESET : Führt einen Hardware Reset aus und bootet JABBA neu. Dabei wird die aktuelle Konfiguration verworfen und mit jener im residenten (nicht-flüchtigen) Speicher gestartet.

SAVE : speichert die aktuelle Konfiguration in den residenten Speicher.

Anmerkung: 'SAVE' ist nur nötig, falls nicht jede Konfigurationsänderung automatisch den residenten Speicher beschreibt. Dies kann vom Programmierer im File JabbaCfg.hpp beeinflusst werden, ist jedoch standardmässig aktiviert.

DEFAULT : Lädt die einprogrammierte Defaultkonfiguration. Diese kann danach mit dem Befehl 'SAVE' in den nicht-flüchtigen Speicher geschrieben werden.

6.3 Systemstart mit Defaultkonfiguration

Für den ersten Start von JABBA empfiehlt sich, die Hardware vom Netzwerk zu trennen und den Failsafe Jumper 1 (Abb. 3, “Jumper auf JABBA Hardware,” auf Seite 9) zu setzen. In diesem Falle startet JABBA mit der einprogrammierten Defaultkonfiguration.

Die einprogrammierte Defaultkonfiguration kann auch jederzeit mit dem Befehl 'DEFAULT' in den Speicher gerufen werden.

6.4 Konfiguration über die serielle Schnittstelle

Die Konfiguration von JABBA über die serielle Schnittstelle kann über ein Terminalprogramm vom PC oder VT100 Terminal geschehen. Die serielle Schnittstelle wird von JABBA fortwährend abgehört. Um in den Konfigurationsmodus zu gelangen, ist die Eingabe eines speziellen Tokens nötig.

Nach Eingabe von '%%CONFIG%%' befindet sich JABBA im Konfigurationsmodus. Dies wird mit '%%CONFIG%% A' bestätigt.

Jetzt kann jede Konfigurationsvariable abgerufen oder verändert werden. Durch alleinige Eingabe des Namens, bspw. 'IPADDR' antwortet JABBA mit 'IPADDR A 172.24.113.10'. Gesetzt wird eine Variable durch Angabe des Namens und des Wertes in Anführungs- und Schlusszeichen. So setzt der Befehl 'IPADDR "172.24.115.47"' die IP Adresse auf besagten Wert. Jede Konfigurationsänderung wird mit 'VariablenName A' bestätigt. Die Syntax

der Konfiguration auf der seriellen Schnittstelle befolgt den Mettler internen MTSICS Standard für Produktionssoftware. Mit der Eingabe von '%%END%%' wird der Konfigurationsmodus wieder verlassen.

Anmerkung: JABBA sendet kein Echo der eingegebenen Buchstaben. Es empfiehlt sich für die Konfiguration von JABBA das lokale Echo des Terminals zu aktivieren.

6.4.1 Anmerkung für den Programmierer

Alle Befehle sind als String im File JabbaCfg.hpp deklariert. Der Start und End-Token für den Konfigurationsmodus ist in Jabba.h deklariert.

6.5 Konfiguration über Ethernet

Eine Systemfunktion übernimmt die Konfiguration von JABBA über die Ethernet Schnittstelle. Diese kann entweder über den Browser oder via ein Java-Applet aufgerufen werden. Dies geschieht mittels Aufruf von
http://jabba/sys/server-config?TOKEN[=VALUE]

Gültige Token sind alle Variablennamen der Netzwerk- und Seriellen Schnittstelle aus “Konfigurationsvariablen und deren Defaultwerte” auf Seite 22 sowie die Befehle ‘SAVE’ und ‘RESET’ aus “Zusätzliche Befehle” auf Seite 22. Aus Sicherheitsgründen wurde jedoch die Konfiguration der MAC-Addresse über Ethernet deaktiviert.

Der letzte Teil [=VALUE] ist optional. Wird dieser weggelassen, gibt JABBA den aktuellen Wert der Variablen zurück. Bei einem gültigen Konfigurationsaufruf antwortet JABBA mit HTTP Status Code 203. Erfolgt die Konfiguration über den Browser (nicht via Applet), so kann dies u.U. zu einer Fehlermeldung führen.

Achtung: Der aktuelle Wert tritt erst bei einem Neustart von JABBA in kraft!

6.6 JABBA Konfiguration für den Programmierer

Die Konfiguration wird JABBAintern in zwei verschiedenen structs aufbewahrt. Diese sind (struct SERIAL_CONFIG) aSerialConfig, definiert in JabbaCFG.hpp, und (struct NETWORK_CONFIG) aNetworkConfig, definiert in /drv/pcconfig.hpp. In den Strukturen gibt es für jede Variable ein Feld, das dessen Wert beinhaltet.

Um die Konfiguration über einen Restart zu erhalten, wird von beiden Strukturen eine Kopie im nicht-flüchtigen Speicher abgelegt.

6.6.1 Initialisierung der Konfiguration beim Systemstart

Beim Systemstart erfolgt folgender Ablauf:

1. Das Betriebsystem startet den JabbaSrv Task, der als erstes die Konfigurationsroutinen aufruft.
2. Die Konfigurationsroutinen überprüfen, ob der FailSafe Jumper 1 gesetzt ist und initialisieren allenfalls die einprogrammierte Defaultkonfiguration.
3. Ist der Jumper nicht gesetzt, so wird die Konfiguration im nicht-flüchtigen Speicher überprüft. Steht in der 1. Speicherzelle NACH der Konfiguration (ermittelt via sizeof(struct)) ein ‘@’, so kann angenommen werden, das im Speicher eine gültige Konfiguration abgelegt worden ist. Diese wird sodann vom nicht-flüchtigen Speicher zurück ins RAM geschrieben.

4. Als letzter Punkt wird die serielle Schnitstelle, der NIC und der IP-Stack initialisiert. Die Initialisierungsroutinen greifen alle auf die in den Strukturen gespeicherten Werte zu.

Der Ablauf der Systeminitialisation wird im in Abbildung 1 graphisch dargestellt.

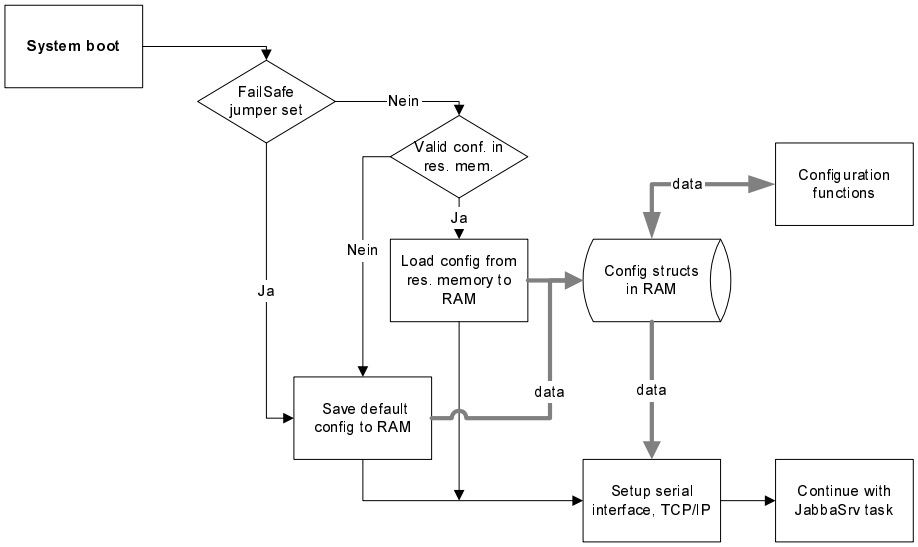


FIGURE 9. Konfiguration von JABBA beim Systemstart

Achtung: Die Konfigurationsdaten im nicht-flüchtigen Speicher werden NICHT nach Konsistenz überprüft. Das ‘@’ am Ende der Einträge garantiert nur, dass die Konfiguration vollständig kopiert wurden.

6.6.2 Änderung der Konfiguration im laufenden Betrieb

Wird während Laufzeit die Konfiguration verändert, so werden die Konfigurationsvariablen der Struktur im RAM aufdatiert. Standardmässig wird die Konfiguration nach jeder Veränderung im nicht-flüchtigen Speicher aufdatiert. Dieses Verhalten ist über ein #define in Jabba.hpp beeinflussbar.

Wird die automatische Speicherung deaktiviert, so muss dies nach erfolgter Konfiguration mit dem Befehl 'SAVE' nachgeholt werden.

Die veränderten Einstellungen treten jedoch erst nach einem Neustart in Kraft. Der IP-Stack wird erst beim Neustart neu initialisiert, sodass die veränderten Werte Einfluss nehmen können.

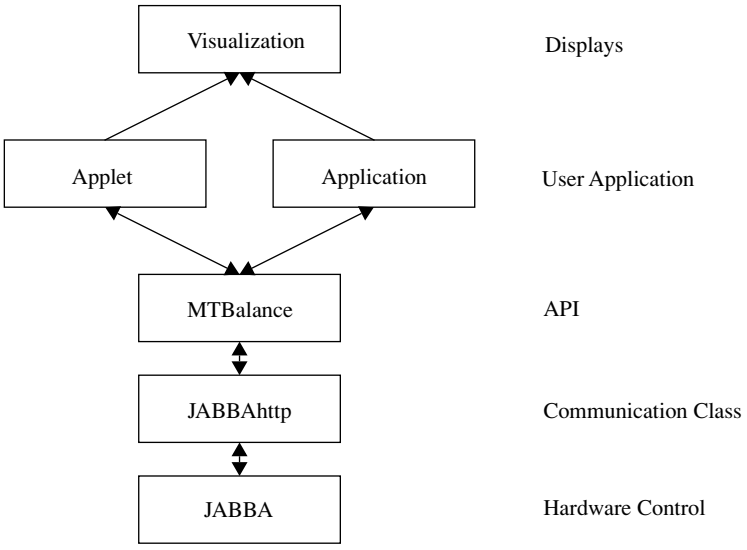
7.0 Java Software

7.1 Übersicht

Die Java-Software besteht aus folgenden Teilen:

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

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



7.2 APIs

7.2.1 JABBAhttp

Diese Klasse stellt die einfachsten Methoden zur Kommunikation mit Jabba zur verfügung:

```
// JABBAhttp.java

// Every request to <name> must include the Jabba-URL
// Get the URL with getDocumentBase() or getCodeBase()
// Must include the "/usr"-path to communicate with the
// UserProcess on Jabba

// Methods

// Constructor: empty

public static String http_get(String name)
// Gets content of URL <name> by HTTP
// pre:  <name> must be valid URL
// post: HTTP-page or empty string if error occured

public static String http_post(String name, String data)
// Posts content of <data> to URL <name> by HTTP
// pre:  <name> must be valid URL
// post: HTTP-answer-page or empty string if error occured

public static String http_post(String name, byte[] data)
// Posts content of <data> to URL <name> by HTTP
// pre:  <name> must be valid URL
// post: HTTP-answer-page or empty string if error occured
```

7.2.2 MTBalance

Diese Klasse bildet die Waage nach. Jegliche Kontrollfunktionen müssen über diese Klasse laufen ! Somit wird ein einfaches Anpassen an Waagentypen ermöglicht (Vererbung).

```
// MTBalance.java

// Public Variables

public Double weight;                                   // aktuelle werte
public String unit;
public String state,time;
public boolean valid;
public boolean stable;
public int decPntPos = 0;

// Methods

public MTBalance( URL adr )                             // konstruktor
// Constructor, URL points to Jabba
// pre:   jabba connected to scale
// post:  balance object ready to use
//        all variables zero
```

```
public MTBalance()
// Constructor
// pre:   nothing
// post:  balance object not ready
//        state = "not ready"

public void updateWeight()
// Updates the weight, unit, state & stable variables
// pre:   jabba connected to scale
// post:  weight, unit, state, stable variables updated

public void inquiryBalance()
// Updates the state variable with the inquiry answer
// pre:   jabba connected to scale
// post:  state variable updated

public void setZero()
// Updates the weight, unit, state & stable variables and
// sets scale to zero.
// pre:   jabba connected to scale
// post:  weight = 0, unit, state = "Set Zero" variables updated

public void getTime()
// Updates the time variable with current weighing time
// pre:   jabba connected to scale
// post:  time variable updated

public void setDisplay( String out )
// Updates state variable and sets the scale display to <out>
// pre:   jabba connected to scale
// post:  state variable updated

public void releaseDisplay()
// Updates state variable and lets the scale display show the
// weighing result
// pre:   jabba connected to scale
// post:  state variable updated

public void tare()
// Updates the weight, unit & state variables and sets the scale
// to zero.
// pre:   jabba connected to scale
// post:  weight, unit, state variables updated
```

Implementation: MTBalance wird mit einer URL als Parameter initialisiert. Diese URL ist der Jabba-Server mit der angeschlossenen Waage. Alle Waagendaten werden als public-Variablen gespeichert. Die Daten werden nach den Methodenaufrufen aktualisiert. Die Methoden rufen die doCommand-Methode auf welche den MTSICS-Code an Jabba sendet. Die Antwort wird geparst und die entsprechenden Werte aktualisiert. Zur Kommunikation wird JABBAhttp instanziert.

7.3 Applets

Beim Entwickeln der Applets ist zu beachten dass ein Applet nur Verbindungen auf seinen Host eröffnen können ! Insbesondere im Appletviewer muss auf diese Restriktion

geachtet werden. Java 2 stellt für die meisten Browser im Moment noch ein Problem dar. Diese Klassen werden bis jetzt nicht verwendet.

7.3.1 JApplet

Test für Get & Post der JABBAhttp-Klasse.

Post-Output:	vom Textfenster
Get-Input:	ins Textfenster
URL-Pfad:	Textfeld ganz oben

Implementation: Init generiert das GUI. Weiter wird nur auf Events reagiert und die entsprechenden JABBAhttp-Funktionen aufgerufen.

7.3.2 JBMT

Einlesen des Gewichts und Ausgabe als Text. Es muss eine Waage an Jabba angeschlossen sein.

Implementation: Init generiert das GUI und instanziert MTBalance. In der Run-Methode wird bei Bedarf (Polling=true) die Funktion MTBalance.updateWeight() aufgerufen und das Gewicht angezeigt.

Ansonsten wird nur auf Button-Events reagiert:

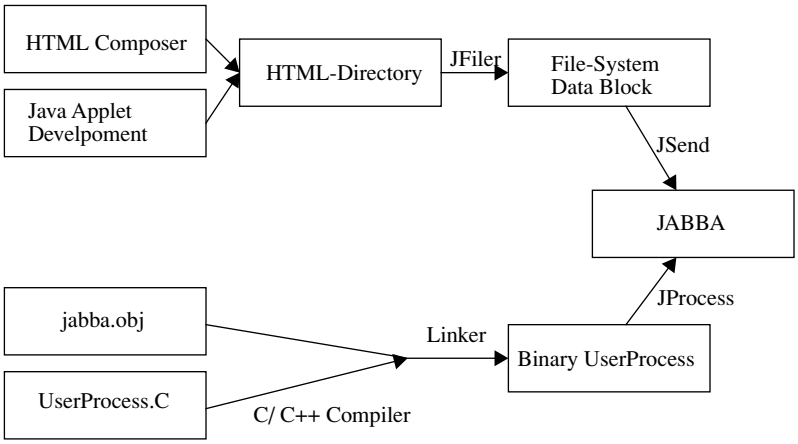
SetZero:	ruft MTBalance.setZero auf
GetWeight:	macht MTBalance.updateWeight()
InfoBalance:	fragt den Waagentyp ab (MTBalance.inqiry())
Change Disp:	ändert das Waagendisplay (MTBalance.changeDisplay(“Jabba Control”))

7.3.3 JBNuss

Anzeige des Gewichts als Menge von Objekten gleicher Masse. Es muss eine Waage angeschlossen sein.

Implementation: Der Messteil ist analog zu JBMT aufgebaut. Die Visualisierung erfolgt in der Klasse NuesslePot, die Charakteristik der “Nüsslein” in der Nuessle-Klasse. Die Visualisierung erfolgt nur alle 0.5s damit Messfehler einfacher zu filtern sind. Das Fallen der Kugeln wird ganz rudimentär nachgebildet. Gleiten sie aufeinander ab so geschieht das auf einer Geraden und nicht auf dem Kreisbogen. Einige “Nachzeichenfehler” werden bewusst nicht ausgeschaltet um ein lebendiges Bild zu erhalten (Ausschalten ist effektiv möglich !).

7.4 File-Handling für Uploads



7.4.1 JSend

Sendet den von JFiler generierten Datenblock an Jabba.

Aufruf: java JSend <ip> <file>
Default: ip = 172.24.113.9 file = to.jabba (Output von JFiler)
Output: Sent XXX bytes OK

Implementation: Öffnet File, liest es ein und sendet es an die IP-Adresse mit “POST / sys/data-upload” an den Port 80 (Webserver Jabba). Es wird die Post-Funktion der JABBAhttp-Klasse verwendet.

7.4.2 JProcessMot

Sendet das Motorola S-Record-File des UserProcess an Jabba.

Aufruf: java JProcess <ip> <file>
Default: ip = 172.24.113.9 file = jabbausr.abs
Output: Sent XXX bytes OK

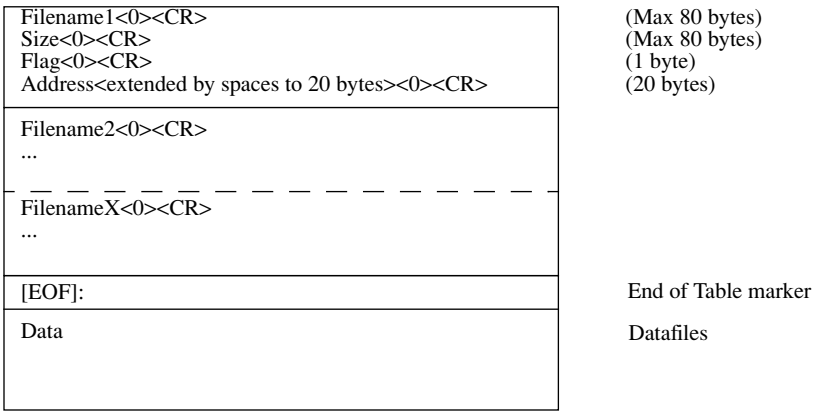
Implementation: Öffnet File, liest es ein und sendet es an die IP-Adresse mit “POST / sys/process-upload” an den Port 80 (Webserver Jabba). Es wird die Post-Funktion der JABBAhttp-Klasse verwendet.

7.4.3 JFiler

Hängt alle Files im Directory zu einem Block zusammen und generiert Header.

Aufruf: java JFiler <directory> <offset>
Default: directory zwingend anzugeben
offset = 0, wird zur Adresse des Files dazugezählt
Output: Liste aller Dateien im Header to.jabba Datei

Implementation: Das angegebene Directory wird rekursiv durchsucht und alle Dateien in eine Temporärdatei aneinandergehängt. Die Dateinamen, inklusive Pfad, und -Längen werden in Arrays gespeichert. Anschliessend wird die Datei to.jabba eröffnet. Die Headerdaten werden mitsamt der berechneten Adresse in diese Datei geschrieben und die vorher generierte Temporärdatei an den Header angehängt.



8.0 Flash Programmierung

Das Datenflash wird in drei Sektionen unterteilt:

- HTML-Daten
- UserProzess
- Parameter

Für alle drei Sektionen werden eigene Routinen verwendet. Diese berücksichtigen die Blockstruktur des Bausteins. Für jede Sektion gibt es eine Löschfunktion und eine Schreibfunktion, sowie eine Funktion zum Abfragen der Anzahl bereits geschriebener Bytes. Vor dem Schreiben müssen die Flash-Speicher zwingendermassen gelöscht werden ! Sie enthalten dann den Wert FFhex.

Funktionen zur Flashprogrammierung: Siehe JabbaFlash.hpp

8.1 Implementation

8.1.1 Flash_<Sektion>_Clear

Ruft Flash_Clear auf mit der entsprechenden Blockadresse. Darin wird in drei Write-Zyklen der Clear-Befehl gegeben. Anschliessend wird gecheckt ob der Vorgang fehlerfrei beendet wurde. Wenn ja wird TRUE zurückgegeben.

8.1.2 Flash_<Sektion>_Write

Ruft Flash_Write_Block auf mit entsprechenden Adressen. Der Schreibzähler der Sektion wird bei erfolgreichem Schreiben ebenfalls erhöht. Die Flash_Write_Block-Routine prüft auf Blocküberläufe und schreibt Word um Word ins Flash (Flash_Write_word). Wurde zuletzt ein Byte geschrieben so wird das letzte Word geladen, das erste neue Byte dazugenommen und als Word zurückgeschrieben. Dann wird Word um Word programmiert. Bleibt am Schluss ein Byte übrig, so wird dieses als Low-Byte ins nächste Flash-Wort geschrieben. Das höhere Byte muss dabei FFhex sein, da das Flash nur von 1 auf 0 programmieren kann. Dies ist nötig da nicht alle ankommenden Daten Word-Aligned sind.

8.1.3 Flash_Ident

Wird von Flash_Clear zuerst aufgerufen um festzustellen ob das richtige Flash vorhanden ist. Es werden Manufacturer- & Revision-Code geprüft.

9.0 File System

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. Durch das generieren des Datenblocks auf dem PC ist diese Funktion sehr klein.

Format des Datenblocks:

Filename1<0><CR> Size<0><CR> Flag<0><CR> Address<extended by spaces to 20 bytes><0><CR>	(Max 80 bytes) (Max 80 bytes) (1 byte) (20 bytes)
Filename2<0><CR> ... FilenameX<0><CR> ...	
[EOF]:	End of Table marker
Data	Datafiles

9.1 Funktion fHttp_Get_File(name, start_ptr, size)

Die Filetabelle wird nach dem richtigen Filenamen durchsucht und die entsprechenden Daten zurückgegeben. Falls die Daten korrupt sein sollten, bricht der Suchvorgang nach 300 Dateinamen ab. Rückgabe in diesem Fall: Null-Pointer.

Diese Funktion ist Teil von httpd.cpp.

10.0 UserProcess Upload

Um das Jabba-Tool perfekt auf jedes Gerät anpassen zu können ist ein UserProzess vorgesehen. Dieser UserProzess wird speziell für die zu steuernde Maschine entwickelt. Programmiert wird er in C/ C++ und dann crosscompiliert. Nach dem Linken erfolgt der Upload mit Hilfe von JProcess. Nach einem Neustart wird, sofern der 2. Jumper nicht gesetzt ist, der UserProzess gestartet. Ansonsten wird der Default-UserProzess verwendet.

10.1 Programmierung

Die Hauptschleife des Prozesses muss die Kontrolle des angeschlossenen Gerätes sowie die IPC Auswertung beinhalten. Dazu steht ein Template-File zur Verfügung. Siehe Kapitel 5, UserProcess.

Für die I/O Geräte können die Standardtreiber der Tigris-Software verwendet werden. Variablen können bis zu 1kByte angelegt werden, darüber hinaus müssen dynamische Allozierungen verwendet werden. Dies folgt aus dem Makefile und der daraus resultierenden Speicheraufteilung.

10.1.1 IPC

Der UserProzess erhält vom Webserver jegliche Verbindungsdaten. Diese dürfen nicht verändert werden. Sie sollen aber für einen allfälligen Verbindungsaufbau vom UserProzess her bekannt sein. Der UserProzess darf den Contentpointer verändern, muss aber den alten Content vorher freigeben !

Datenstruktur: siehe Kapitel 3.3, Jabba Webserver

10.2 Compilieren, Linken

Mit dem CPP68000 Compiler wird der Code crosscompiliert. Fürs Linken muss das Jabba-Makefile verwendet werden. Es lädt die symbolischen Links zur Standardsoftware und bestimmt die Destinations-Adresse für den Code. Ausserdem werden die Variablen ans richtige Ort im RAM gemappt. Diese dürfen eine Länge von 1kByte nicht überschreiten (ausser dynamische Allozierung) ! Der Code wird ab 608000Hex ins Flash gemappt, danach folgen die Konstanten. Die Variablen werden entsprechend dem Jabba.Obj-File im RAM angeordnet. Stellen Sie desswegen sicher dass die Software auf der Jabba Box mit derjenigen im Jabba.Obj-File übereinstimmt !

10.3 Upload auf die Jabba Box

Der Binärcode (UserP.mot) kann nun mit Hilfe des JProcess-Tools auf Jabba geschrieben werden.

Aufruf: java JProcessMot <ip-addr> <filename>

Default: ip = 172.24.113.10

file = UserP.mot

Precondition: Der Default-UserProzess muss aktiviert sein !

Output: Wrote XXXX Bytes successfully oder Fehlerbeschreibung. Danach muss Jabba neu gestartet werden. Beim Neustart wird der 2. Jumper getestet und bei Kurzschluss der Default-Prozess gestartet. Stellen Sie also sicher dass der Jumper entfernt ist !

10.4 Wenn nichts mehr geht ...

Mit Hilfe des 2. Jumpers kann der eigene Prozess nach einem Neustart abgeschaltet werden. Somit ist Jabba frei für einen neuen Versuch.

11.0 Anhang A: Files

11.1 C und C++ Software

/default	(*.cpp & *.hpp -Files)
/app	
JabbaSys	Systemfunktionen
JabbaSrv	TCP-Connection-Verwaltung
JabbaCnf	Config-Funktionen
httpd	HTTP-”Dämon”
JabbaFlash	Funktionen zur Flash-Programmierung
/drv	
Tigris-Treiber	Hardware-Treiber aus Tigris Projekt
/os	
OS68cnf	Prozessliste des OS

11.2 Java Software

/sw_java	
/utilities	(*.class & *.java -Files)
JFiler	File-Tool
JSend	Daten-Upload
JProcessMot	Prozess-Upload
JABBAhttp	Kommunikationsklasse
/balance	(*.class & *.java -Files)
JApplet	Testapplet
JBMT	Simples Wägeapplet
JBNuss	Beispielvisualisierung
MTBalance	Waageklasse
JABBAhttp	Kommunikationsklasse

11.3 HTML-Demoseite

/html	
index.html	Startseite
/pics	
diverse	Bilder
/java	
class-files	Applets