appliedinformatics

# Web Browser basierte Benutzerschnittstellen für Linux-basierte Embedded Systeme

Günter Obiltschnig
Applied Informatics Software Engineering GmbH
St. Peter 33
9184 St. Jakob im Rosental
Austria
guenter.obiltschnig@appinf.com

## 1 Einleitung

Aufgrund seiner hervorragenden Netzwerkfähigkeiten wird Embedded Linux als Betriebssystem gerne für Systeme verwendet, welche an ein TCP/IP basiertes Netzwerk (über Ethernet, WLAN, GPRS, etc.) angebunden werden sollen. Solche Systeme werden in vielen Fällen mit einem integrierten Web Server ausgestattet, mit dem das Gerät über das Netzwerk, oder sogar über das Internet, gesteuert, überwacht und konfiguriert werden kann. Es gibt verschiedene Möglichkeiten, einen derartigen Web Server zu implementieren.

Viele Linux Distributionen für den Einsatz in Embedded Systemen sind bereits mit einem einfachen, ressourcenschonenden HTTP Server, wie z. B. Boa[1] oder thttpd[2] ausgestattet. Neben statischen HTML Seiten und Bildern können diese Web Server über sogenannte CGI Programme (Common Gateway Interface – eine standardisierte Programmschnittstelle) dynamische HTML Seiten generieren. CGI Programme können sowohl in Scriptsprachen – im einfachsten Fall kann ein CGI Programm als Unix Shell Script geschrieben werden – als auch z. B. in C oder C++ entwickelt werden. Der Vorteil von CGI Programmen, speziell wenn sie in einer Scriptsprache geschrieben werden – ist deren einfacher Test. Im Falle einer Scriptsprache reicht es aus das Script zu ändern und das Ergebnis kann sofort über ein Neuladen der Seite im Browser getestet werden. Eine CGI-basierte Lösung hat jedoch auch große Nachteile. Für jeden Seitenaufruf muss ein Prozess gestartet und ein Programm ausgeführt werden. Dies stellt, speziell bei kleineren Systemen, eine nicht unwesentliche Systembelastung dar. Ein weiterer Nachteil betrifft die Art und Weise, wie die in der HTML Seite darzustellenden, bzw. zu bearbeitenden Daten das CGI Programm erreichen. Sollen lediglich Konfigurationsparameter des Systems dargestellt und geändert werden, so ist dies fallweise noch einfach, da diese Parameter üblicherweise in einer oder mehreren Dateien im Dateisystem liegen. Aus diesen Dateien werden die Parameter gelesen und für die Darstellung in einer HTML Seite formatiert. Auch das Zurückschreiben geänderter Parameter ist noch relativ einfach zu implementieren. Die Konfigurationsdateien werden neu geschrieben, und die betroffenen Prozesse (oder sogar das ganze System) werden neu gestartet. Oft ist es aber so, dass dynamische Daten direkt aus der Applikation, wie z. B. Messwerte, Performance-Daten, usw. über den Web Server bereitgestellt werden sollen, oder eine Applikation sogar über ein Web Interface gesteuert werden soll. Hier muss also der Applikationsprozess mit den CGI Programmen kommunizieren. Dies verursacht erheblichen Aufwand, sowohl zur Entwicklungszeit als auch zur Laufzeit.

---

[1] http://www.boa.org/
[2] http://www.acme.com/software/thttpd/

Eine Alternative zum Einsatz eines eigenständigen Web Server Prozesses ist, den Web Server direkt als eigenen Thread innerhalb der eigentlichen Applikation laufen zu lassen. Offensichtliche Vorteile einer solchen Lösung sind einerseits der einfache, direkte Zugriff auf die dynamischen Daten der Applikation zwecks Visualisierung auf einer HTML Seite, anderseits die Performance, da zur Generierung von dynamischen HTML Seiten keine Prozesse gestartet werden müssen.

# 2 HTTP – Hyper-Text Transfer Protocol

Web Browser und Web Server kommunizieren untereinander über das Hyper-Text Transfer Protocol (HTTP). In seinen Grundzügen ist HTTP sehr einfach strukturiert und leicht verständlich. Allerdings ist das Protokoll in seiner Gesamtheit doch einigermaßen komplex. Die Spezifikation von HTTP findet man in RFC 2616[3] – sie ist 176 Seiten lang. Sofern man nicht selbst einen vollständigen HTTP Server entwickeln möchte, reichen ein paar Grundlagen aus, welche folgend erläutert werden.

Grundsätzlich basiert das HTTP Protokoll auf einem Anfrage-Antwort Schema. Der Web Browser (oder ein anderer Client) sendet eine Anfrage an den Server, woraufhin der Server eine Antwort zurücksendet. HTTP ist Text-basiert, und daher sehr einfach zu testen und debuggen. Im einfachsten Fall kann man einen HTTP Server mit einem Telnet-Programm testen. Für die folgenden Beispiele wird angenommen, dass ein Linux Entwicklungsboard mit einem HTTP Server über die IP Addresse 192.168.1.201 erreichbar ist. Der HTTP Server läuft auf TCP Port 80. Von einer Linux Shell (oder auch einer Windows Shell) aus, kann man zu Testzwecken folgendes Kommando eingeben:

```
$ telnet 192.168.1.201 80
Trying 192.168.1.201...
Connected to 192.168.1.201.
Escape character is '^]'.
GET /index.html HTTP/1.0 (2x return)

HTTP/1.0 200 OK
Date: Wed, 10 Sep 2008 15:41:32 GMT
Server: Boa/0.94.14rc21
Accept-Ranges: bytes
Connection: close
Content-Length: 689
Last-Modified: Thu, 03 Jul 2008 10:44:11 GMT
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
...
```

*Beispiel 1: HTTP 1.0 Anfrage über Telnet*

Wichtig ist, nach dem `GET / HTTP/1.0` zwei mal auf die Return-Taste zu drücken. Dies teilt dem Server mit, dass der Request vollständig ist, und er diesen verarbeiten kann. Daraufhin wird der HTTP Server normalerweise ein `HTTP/1.0 200 OK`, gefolgt von Metadaten und dem HTML Dokument zurücksenden und die Verbindung schließen.

Für das Beispiel wurde Version 1.0 von HTTP verwendet. Aktuell ist allerdings schon seit geraumer Zeit Version 1.1. Version 1.0 funktioniert zwar weiterhin, es kann allerdings bei Servern die mehrere Websites hosten (z. B. bei Hosting-Providern), zu Problemen kommen, da nicht klar ist, welche Website gemeint ist. Bei HTTP Version 1.1 sieht das Beispiel folgendermaßen aus:

---

[3] http://www.w3.org/Protocols/rfc2616/rfc2616.html

```
GET /index.html HTTP/1.1
Host: 192.168.1.201 (2x return)

HTTP/1.1 200 OK
Date: Wed, 10 Sep 2008 15:44:21 GMT
Server: Boa/0.94.14rc21
Accept-Ranges: bytes
Connection: Keep-Alive
Keep-Alive: timeout=10, max=1000
Content-Length: 689
Last-Modified: Thu, 03 Jul 2008 10:44:11 GMT
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
...
```

*Beispiel 2: HTTP 1.1 Anfrage über Telnet*

Der wesentliche Unterschied neben HTTP/1.1 ist die zusätzliche Angabe eines Host Feldes im Request. Dies ist bei HTTP 1.1 vorgeschrieben. Ein weiterer Unterschied ist, dass der Server die Verbindung nicht unmittelbar nach der Übertragung des HTML Dokuments schließt, sondern noch für einige Zeit (10 Sekunden im Beispiel) offen lässt. Dies ermöglicht es, auf einer Verbindung mehrere Anfragen zum Server zu senden (sog. *persistent connections*). Erfolgt in dieser Zeit allerdings keine neue Anfrage, schließt der Server die Verbindung. Da der Overhead für das Öffnen einer TCP Verbindung relativ hoch ist, und normalerweise zur Darstellung einer Seite im Web Browser mehrere Anfragen zum Server gesendet werden müssen (HTML Dokument, Stylesheet, eingebettete Bilder, etc.), ermöglicht dieses Verfahren eine wesentlich schnellere Darstellung von Web-Seiten bei gleichzeitig verringerter Server-Last.

HTTP kennt verschiedene Arten von Anfragen (sogenannte Request-Methoden). Eine davon, GET, wurde im Beispiel verwendet. Daneben gibt es noch zwei weitere wichtige, nämlich POST und HEAD. POST wird verwendet, wenn größere Datenmengen (z. B. Hochladen einer Datei), bzw. Daten zum Server gesendet werden sollen, die am Server eine Veränderung (z. B. in einer Datenbank) hervorrufen. Zwar könnte man kleinere Daten auch mit GET übertragen, es ist jedoch guter Stil, mit GET nur Anfragen zum Server zu senden die jedes Mal das identische Ergebnis liefern (idempotente Anfragen). Dies ermöglicht ein Zwischenspeichern von Daten z. B. im Browser-Cache oder auf einem Caching-Server. Dokumente, die als Antwort auf ein POST Anfrage gesendet werden, werden von einem Cache üblicherweise nicht zwischengespeichert. Interessant ist noch HEAD. Der Server macht bei einem HEAD prinzipiell das selbe wie bei einem GET, er liefert jedoch das Dokument nicht an den Client zurück, sondern nur dessen Metadaten.

```
HEAD /index.html HTTP/1.1
Host: 192.168.1.201

HTTP/1.1 200 OK
Date: Sat, 13 Sep 2008 16:09:20 GMT
Server: Boa/0.94.14rc21
Accept-Ranges: bytes
Connection: Keep-Alive
Keep-Alive: timeout=10, max=1000
Content-Length: 689
Last-Modified: Thu, 03 Jul 2008 10:44:11 GMT
Content-Type: text/html
```

*Beispiel 3: HTTP HEAD Anfrage über Telnet*

Dies wird von einem Web Browser dazu benutzt um festzustellen, ob ein Dokument, welches der Browser bereits im Cache hat, am Server verändert wurde. Dies kann z. B. anhand des Date (oder ETag) Feldes festgestellt werden. Nur im Falle eines geänderten Dokuments wird der Browser das Dokument mit GET neu vom Server laden.

Betrachtet man die Struktur einer HTTP Anfrage, bzw. Antwort, so fällt auf, dass beide aus jeweils drei Teilen bestehen. Der erste Teil (erste Zeile) enthält bei einer Anfrage die Request-Methode (GET, POST, HEAD, etc.), den Namen, bzw. Pfad der Resource, welche vom Server gesendet werden soll, sowie die Version des HTTP Protokolls (1.0 oder 1.1). Bei einer Antwort enthält die erste Zeile als erstes die HTTP Version, gefolgt von einem Status Code und einem Status Text. Wichtige Status Codes sind 200 (OK), 404 (Not Found) und 500 (Internal Server Error). Auf die erste Zeile folgen die sogenannten Header-Daten. Diese können die Anfrage oder Antwort näher spezifizieren (z. b. erwartete Dokumenten-Typen bei einer Anfrage, bzw. Typ des gesendeten Dokuments bei einer Antwort), Informationen über den Client an den Server übertragen (z. B. Name und Version des Clients) und umgekehrt, sowie das gesendete Dokument näher beschreiben (Größe, Datum der letzten Änderung, etc.). Pro Zeile wird der Name und Wert eines Feldes übertragen. Die Anzahl der Felder ist nicht beschränkt. Um das Ende der Header-Daten zu kennzeichnen, wird eine leere Zeile gesendet. Zeilenenden werden in HTTP immer mit CR LF gekennzeichnet. Auf die Header-Daten folgt, so vorhanden, das eigentliche Dokument.

Interessant ist noch die Frage, wie man erkennt, dass das gesendete Dokument vollständig gesendet wurde. Dies kann entweder dadurch erfolgen, dass man im Header die Länge des Dokuments (Content-Length) überträgt. Ist die Länge vorab nicht bekannt, was bei dynamisch generierten Daten der Fall sein kann, so gibt es das sogenannte Chunked Transfer Encoding bei HTTP Version 1.1. Hierbei wird das Dokument in mehreren Fragmenten gesendet, wobei vor jedem Fragment dessen Länge gesendet wird. Zu guter Letzt kann man nach der vollständigen Übertragung des Dokuments einfach die Verbindung schließen. Dies sollte allerdings bei HTTP Version 1.1 nicht mehr verwendet werden.

# 3 Common Gateway Interface (CGI)

Ein HTTP Server wie Boa oder thttpd ist darauf ausgelegt, HTML Dokumente, Bilder, usw., welche in einem speziellen Verzeichnis im Dateisystem liegen (z. B. */usr/share/www*), auszuliefern. Nun sind solche Dateien statisch, d. h. sie werden üblicherweise nicht laufend verändert. Wie kann ein solcher Server nun dynamische Daten (z. B. sich laufend verändernde Messwerte, die von einem Gerät erfasst werden) darstellen? Eine Möglichkeit wäre, eine Applikation laufen zu lassen, welche ständig eine HTML Datei mit den letzten Messwerten neu generiert. Dies ist aber ineffizient, und eigentlich auch nicht im Sinne des Web-Gedankens. Ganz abgesehen davon, dass bei Embedded Systemen übliche Flash-Speicher nur eine beschränkte Anzahl von Schreib-Zyklen ermöglichen (eine RAM-Disk könnte Abhilfe schaffen). Eine wesentlich bessere Alternative sind jedoch sogenannte CGI Applikationen. Das Common Gateway Interface[4] (CGI) ist eine (inoffiziell) standardisierte Schnittstelle welche Web Server verwenden, um bei einem Aufruf bestimmter Seiten ein externes Programm zu starten, welches dynamisch ein Dokument (HTML, Bild, etc.) generiert und über den Web Server an den Client sendet. Ein CGI Programm kann in nahezu jeder beliebigen Programmiersprache geschrieben werden – in C oder C++, Perl, Python, oder auch als Shell Script.

Ruft der Web Server ein CGI Programm auf, so werden Informationen über die HTTP Anfrage (Request-Methode, HTTP Version, Header Felder) über Environment Variablen an das CGI Programm übergeben. Das CGI Programm generiert ein Dokument und schreibt es auf die Standard-Ausgabe (*stdout*). Der Web Server sendet das Dokument weiter an den Client.

---

[4] http://hoohoo.ncsa.uiuc.edu/cgi/

Einige nützliche CGI Environment Variablen sind in Tabelle 1 aufgelistet.

| | |
|---|---|
| `SERVER_PROTOCOL` | HTTP Version (z. B. HTTP/1.1) |
| `REQUEST_METHOD` | HTTP Methode (z. B. GET, POST) |
| `QUERY_STRING` | Form-Daten bei GET |
| `REMOTE_ADDR` | IP Adresse des Clients |

*Tabelle 1: CGI Environment Variablen*

Mit dieser Information kann ein einfaches CGI Script geschrieben werden:

```
#! /bin/sh

echo "Content-Type: text/plain"
echo ""

echo "Server Protocol: $SERVER_PROTOCOL"
echo "Request Method:  $REQUEST_METHOD"
echo "Query String:    $QUERY_STRING"
echo "Client Address:  $REMOTE_ADDR"
```

*Beispiel 4: Einfaches CGI Programm als Shell Script*

Wird dieses Script im richtigen Verzeichnis (z. B. */usr/lib/cgi-bin* bei Boa) abgelegt und mit Ausführungsrechten versehen, kann es im Browser aufgerufen werden, wie in Abbildung 1 gezeigt.

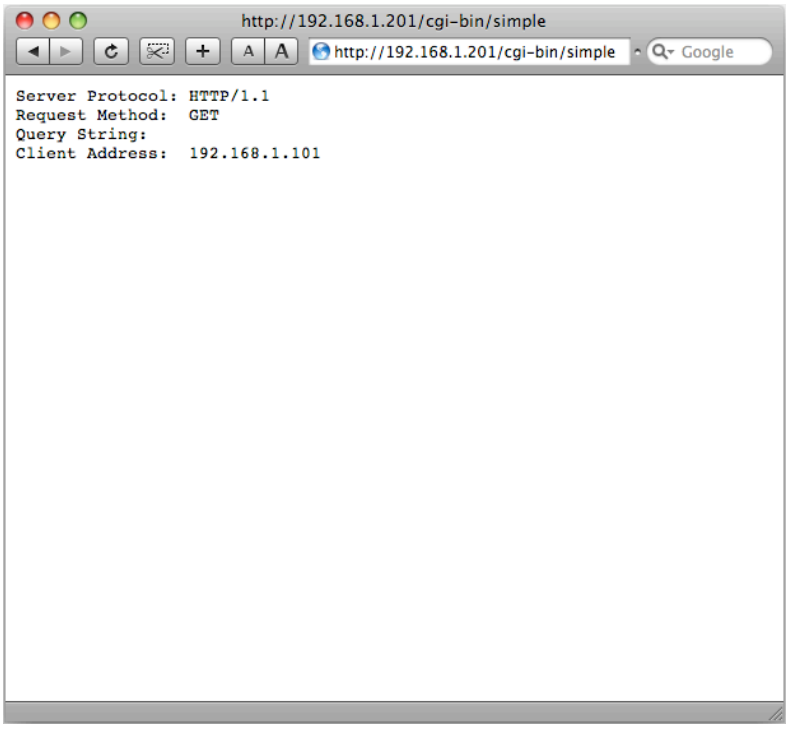


*Abbildung 1: Aufruf eines CGI Scripts*

Dieses sehr einfache CGI Script generiert eine Text-Seite (kein HTML!), zeigt aber dennoch schon gut die Funktionsweise eines CGI Programms. Bevor das CGI Programm noch irgendwelchen Text oder HTML ausgibt, muss es zuerst bestimmte Header Information an den Web Server senden. Der wichtigste Header ist `Content-Type`, welcher den Typ des generierten Dokuments angibt. Bei reinem Text ist es *text/plain*, bei HTML *text/html* und bei einem JPEG-Bild *image/jpeg*. Optionale weitere Header sind `Location`, welches den Web Browser anweist, die Anfrage auf die angegebene URL umzuleiten, sowie `Status`, welches

dem Script ermöglicht, den HTTP Status Code zu setzen. Nach den Headern folgt zwingend eine Leerzeile, und danach das eigentliche Dokument.

Selbstverständlich kann ein CGI Shell-Script auch HTML generieren, wie in folgendem Beispiel gezeigt:

```
#! /bin/sh

hostname=`hostname`
datetime=`date`
uptime=`uptime`
osname=`uname`
osver=`uname -r`

echo "Content-Type: text/html"
echo ""

echo "<html>"
echo "<head>"
echo "<title>System Information</title>"
echo "</head>"
echo "<body>"
echo "<h1>System Information for $hostname</h1>"
echo "<table>"
echo "<tr><th>Date/Time:</th><td>$datetime</td></tr>"
echo "<tr><th>Uptime:</th><td>$uptime</td></tr>"
echo "<tr><th>OS Version:</th><td>$osname $osver</td></tr>"
echo "<tr><th>Hardware:</th><td><pre>"
cat /proc/cpuinfo
echo "</pre></td></tr>"
echo "</table>"
echo "</body>"
echo "</html>"
```

*Beispiel 5: CGI Programm mit HTML Ausgabe*

Das Ergebnis des Aufrufs dieses Scripts ist in Abbildung 2 dargestellt.

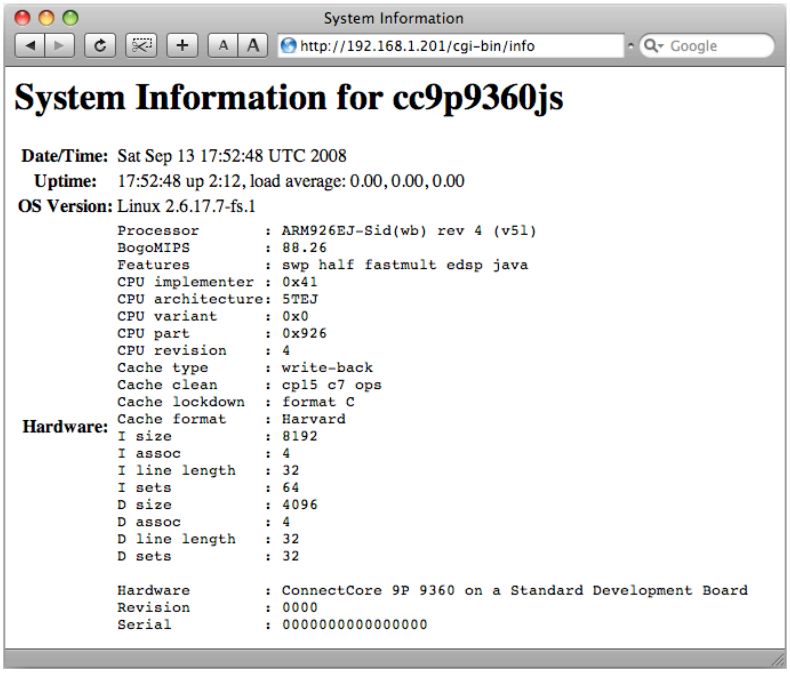


*Abbildung 2: Ein CGI-Script, welches HTML generiert*

Mit einem Shell Script gelangt man allerdings recht schnell an die Grenze des sinnvoll machbaren. Spätestens wenn Formular-Daten ausgewertet werden sollen, sind Alternativen gefragt, da das Parsen der vom Browser übertragenen Formular-Daten in einem Shell Script sehr aufwändig ist. Für C und C++, sowie diverse Script-Sprachen wie Perl, Python, TCL, etc. sind Bibliotheken verfügbar, welche die Implementierung von CGI Programmen wesentlich vereinfachen. Eine interessante Alternative, zumindest bei leistungsfähigeren Embedded-Systemen, ist auch PHP[5]. Bei einfachen HTTP Servern, wie sie bei Embedded Linux üblicherweise verwendet werden, wird der PHP Interpreter über CGI angebunden.

CGI ist allerdings mit drei großen Problemen behaftet. Erstens muss für jeden Aufruf einer Seite ein Programm gestartet werden, was auf weniger leistungsfähigen Microcontrollern eine nicht unerhebliche CPU Belastung darstellt. Dies ist noch schwerwiegender, wenn das CGI Programm in einer interpretierten Script-Sprache implementiert ist – bei jedem Seitenaufruf muss das Script dann neu interpretiert werden.

Zweitens muss ein Weg gefunden werden, wie das CGI Programm an die darzustellenden Daten gelangt. Sind die Daten im Dateisystem vorhanden so ist dies einfach. Sollen jedoch interne Daten einer Applikation angezeigt werden, bzw. sogar eine Applikation über das Web gesteuert werden können, so muss das CGI Programm mit der Applikation kommunizieren. Dies kann auf verschiedene Arten erfolgen, z. B. über gemeinsame Dateien, über Shared Memory oder andere IPC Mechanismen. So etwas korrekt und performant zu implementieren ist jedoch keinesfalls trivial.

Drittens ist es schwierig, Informationen zwischen verschiedenen Seiten-Aufrufen zwischenzuspeichern. Für jeden Aufruf wird ein eigener Prozess gestartet. Ist die Seite fertig generiert, wird der Prozess beendet und alle darin enthaltene Information ist verloren. Der einzige Weg, größere Mengen an Informationen zwischen einzelnen Seitenaufrufen zwischenzuspeichern ist über das Filesystem, bzw. eine Datenbank. Für kleinere Datenmengen (maximal einige hundert Bytes) können HTTP Cookies[6] verwendet werden.

# 4 Eingebettete HTTP Server

Aufgrund der genannten Probleme von CGI ist es eine interessante Alternative, den HTTP Server nicht als eigenen Prozess laufen zu lassen, sondern den HTTP Server direkt in die Applikation einzubinden. Natürlich heißt dies nicht unbedingt, einen vollständigen HTTP Server von Grund auf zu implementieren. Es gibt bereits verschiedene (quelloffene) Bibliotheken, welche einen HTTP Server implementieren, der nur mehr in die Applikation integriert werden muss. Beispiele sind libmicrohttpd[7] oder die Net Bibliothek der POCO C++ Libraries[8]. Besonders die POCO C++ Libraries sind in diesem Zusammenhang interessant, da es einen speziellen Compiler gibt, der es ermöglicht, ähnlich zu PHP, C++ Code direkt in HTML Dateien einzubinden. Diese HTML Dateien werden vom Compiler in C++ Code übersetzt, der direkt in eine Applikation eingebunden werden kann.

Die Vorgehensweise, um einen eingebetteten HTTP Server in die Applikation einzubinden, ist bei den meisten Bibliotheken ziemlich ähnlich. Zuerst muss ein HTTP Server initialisiert und in einem eigenen Thread gestartet werden. Bei der Initialisierung wird dem Server eine Callback Funktion mitgeteilt, welche aufgerufen wird, sobald eine HTTP Anfrage den Server erreicht. In dieser Callback Funktion wird die HTTP Anfrage ausgewertet, sowie eine entsprechende Antwort generiert, welche dann vom HTTP Server an den Client gesendet wird. Ein einfaches Beispielprogramm in C, basierend auf der libmicrohttpd, wird nachfolgend gezeigt.

---

[5] http://www.php.net/
[6] http://en.wikipedia.org/wiki/HTTP_cookie
[7] http://www.gnu.org/software/libmicrohttpd/
[8] http://pocoproject.org/

```c
#include <microhttpd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>

const char* page =
    "<html><head><title>Time Server</title></head>"
    "<body><h1>%s</h1></body></html>";

static int handleRequest(
    void* cls,
    struct MHD_Connection* connection,
    const char* url,
    const char* method,
    const char* version,
    const char* uploadData,
    unsigned int* uploadDataSize,
    void** context)
{
    int ret = MHD_YES;
    struct MHD_Response* response;

    if (*context != NULL)
    {
        char body[1024];
        time_t now;
        time(&now);
        int bodyLength = sprintf(body, page, ctime(&now));
        response = MHD_create_response_from_data(
                        bodyLength, body, MHD_NO, MHD_YES);
        ret = MHD_add_response_header(response,
                  "Content-Type", "text/html");
        ret = MHD_queue_response(connection, MHD_HTTP_OK, response);
        MHD_destroy_response(response);
        *context = NULL;
    }
    else
    {
        static int dummyContext;
        *context = &dummyContext;
    }
    return ret;
}

int main(int argc, char ** argv)
{
    struct MHD_Daemon* httpServer;
    httpServer = MHD_start_daemon(MHD_USE_THREAD_PER_CONNECTION,
                8080,
                NULL,
                NULL,
                &handleRequest,
                NULL,
                MHD_OPTION_END);
    sleep(100000);
    MHD_stop_daemon(httpServer);

    return 0;
}
```

*Beispiel 6: Verwendung von libmicrohttpd in einem C Programm*

Ein äquivalentes Programm, implementiert in C++ mit den POCO C++ Libraries, ist nachfolgend dargestellt.

```
#include "Poco/Net/HTTPServer.h"
#include "Poco/Net/HTTPRequestHandler.h"
#include "Poco/Net/HTTPRequestHandlerFactory.h"
#include "Poco/Net/HTTPServerParams.h"
#include "Poco/Net/HTTPServerRequest.h"
#include "Poco/Net/HTTPServerResponse.h"
#include "Poco/Net/ServerSocket.h"
#include "Poco/DateTime.h"
#include "Poco/DateTimeFormatter.h"
#include "Poco/DateTimeFormat.h"

using namespace Poco;
using namespace Poco::Net;

class TimeRequestHandler: public HTTPRequestHandler
{
public:
    void handleRequest(
              HTTPServerRequest& request,
              HTTPServerResponse& response)
    {
        DateTime now;
        std::string dateTime(DateTimeFormatter::format(now,
                             DateTimeFormat::ASCTIME_FORMAT));

        response.setChunkedTransferEncoding(true);
        response.setContentType("text/html");

        std::ostream& ostr = response.send();
        ostr << "<html><head><title>Time Server</title>";
        ostr << "<body><h1>" << dateTime << "</h1></body></html>";
    }
};

class TimeRequestHandlerFactory: public HTTPRequestHandlerFactory
{
public:
    HTTPRequestHandler* createRequestHandler(
                            const HTTPServerRequest& request)
    {
        return new TimeRequestHandler;
    }
};

int main(int argc, char** argv)
{
    ServerSocket svs(8080);
    HTTPServer srv(
        new TimeRequestHandlerFactory,
        svs,
        new HTTPServerParams);
    srv.start();
    Thread::sleep(100000);
    srv.stop();

    return 0;
}
```

*Beispiel 7: Verwendung des POCO HTTP Servers in einem C++ Programm*

Trotz der unterschiedlichen Programmiersprachen und verwendeten Bibliotheken erkennt man Gemeinsamkeiten. In der `main` Funktion wird jeweils der HTTP Server initialisiert. Während bei der C Implementierung ein Zeiger auf eine Handler-Funktion (`handleRequest`) an den Server übergeben wird (diese Funktion wird bei Eintreffen einer Anfrage aufgerufen), wird bei der C++ Implementierung ein Zeiger auf ein Factory-Objekt mitgegeben. Bei eintreffen eines Requests wird über diese Factory (Methode `createRequestHandler`) ein `TimeRequestHandler` Objekt erzeugt, welches dann in der Methode `handleRequest` die HTTP Anfrage bearbeitet und eine Antwort generiert.

Das Erstellen von HTML aus C oder C++ heraus kann etwas mühsam sein - zumindest leidet die Lesbarkeit des Codes wenn sehr viele String-Literale mit HTML Fragmenten im Quellcode auftauchen. Auch muss immer wieder derselbe Code geschrieben werden, wenn eine Applikation mehrere verschiedene HTML Seiten anbieten will (was ja durchaus als der Normalfall anzusehen ist).

Um dieses Problem zu lösen gibt es als Teil der POCO C++ Libraries einen sogenannten Page Compiler. Dieser ermöglicht es, HTML und C++ Code in einer Datei zu kombinieren - ähnlich wie man es von PHP her kennt, wo HTML und PHP Code nebeneinander stehen. Der Page Compiler übersetzt ein HTML Dokument (mit eingebettetem C++ Code) in C++ Code (Header-Datei und Implementierungsdatei), der dann zur Applikation gebunden werden kann. Beispielsweise würde aus folgender Datei Code für eine C++ Klasse erzeugt werden, die weitgehend identisch zur `TimeRequestHandler` Klasse im vorigen Beispiel ist.

```
<%@ page class="TimeRequestHandler" %>
<%!
#include "Poco/DateTime.h"
#include "Poco/DateTimeFormatter.h"
#include "Poco/DateTimeFormat.h"

using namespace Poco;
%>

<%
    DateTime now;
    std::string dateTime(DateTimeFormatter::format(now,
                         DateTimeFormat::ASCTIME_FORMAT));
%>

<html>
<head>
<title>Time Server</title>
</head>
<body>
<h1><%= dateTime %></h1>
</body>
</html>
```

*Beispiel 8: Einbettung von C++ in HTML mit dem POCO PageCompiler*

Der große Vorteil eines eingebetteten HTTP Servers ist, dass man zum Generieren der HTML Seiten jederzeit Zugriff auf die gesamten internen Daten sowie den Zustand der Applikation hat. Dies macht es sehr einfach, Daten der Applikation über einen Browser zu visualisieren, bzw. auch Kommandos an die Applikation zu senden.

Ein Nachteil der meisten eingebetteten HTTP Server ist, das diese keine Dateien aus dem Dateisystem (z. B. statische HTML Dateien, Bilder oder Stylesheets) direkt ausliefern können. Dafür muss erst ein spezieller Request Handler geschrieben werden, was aber nicht allzu aufwändig ist.

# 5 HTML Formulare

Sollen auf einer Web Seite nicht nur Daten angezeigt werden, sondern sollen auch z. B. Konfigurationsparameter des Embedded Systems verändert oder eine Applikation gesteuert werden, so müssen HTML Formulare eingesetzt werden. Es gibt insgesamt drei verschiedene Möglichkeiten, wie ein Web Browser Formulardaten an den Server übertragen kann:

- URL-codiert, als Teil der URL (Query String) bei GET
- URL-codiert im Datenteil der Anfrage bei POST
- Als MIME Multipart[9] Dokument bei POST (Datei Upload)

Auf die Details der einzelnen Methoden soll hier nicht weiter eingegangen werden. Das Dekodieren von Formulardaten aus einer HTTP Anfrage ist nicht ganz trivial. Sowohl bei CGI Programmen, als auch bei eingebetteten HTTP Servern verwendet man daher Bibliotheken, die Funktionen zur Verarbeitung von Formulardaten bereitstellen.

Als letztes Beispiel soll eine Applikation gezeigt werden, welche über einen Web Browser gesteuert werden kann. Um das Beispiel kurz und verständlich zu halten, implementiert die Applikation lediglich einen Zähler, der in einem bestimmten Intervall hochgezählt wird. Das Intervall kann über ein HTML Formular verändert werden. Außerdem kann der Zähler über einen Button auf der HTML Seite zurückgesetzt werden. Das Beispiel zeigt auch einen recht interessanten Trick, um mehrere unabhängige Buttons auf einer Web Seite zu platzieren. Für jeden Button wird ein eigenes Formular erstellt, wobei das Formular ein verstecktes (*hidden*) Feld enthält, welches die Aktion des Buttons spezifiziert. Dies ermöglicht es, einfache Submit-Buttons zu verwenden, ohne JavaScript Code schreiben zu müssen.

```
#include "Poco/Net/HTTPServer.h"
#include "Poco/Net/HTTPRequestHandler.h"
#include "Poco/Net/HTTPRequestHandlerFactory.h"
#include "Poco/Net/HTTPServerParams.h"
#include "Poco/Net/HTTPServerRequest.h"
#include "Poco/Net/HTTPServerResponse.h"
#include "Poco/Net/HTTPServerParams.h"
#include "Poco/Net/ServerSocket.h"
#include "Poco/Net/HTMLForm.h"
#include "Poco/Timestamp.h"
#include "Poco/DateTimeFormatter.h"
#include "Poco/DateTimeFormat.h"
#include "Poco/Exception.h"
#include "Poco/NumberParser.h"
#include "Poco/Timer.h"
#include "Poco/Mutex.h"

using namespace Poco;
using namespace Poco::Net;

class Counter
{
public:
    enum
    {
        DEFAULT_INTERVAL = 1000
    };

    Counter():
        _value(0),
```

[9] http://www.faqs.org/rfcs/rfc2387.html

```
        _timer(0, DEFAULT_INTERVAL)
    {
        _timer.start(TimerCallback<Counter>(
            *this, &Counter::onTimer));
    }

    long value() const
    {
        FastMutex::ScopedLock lock(_mutex);
        return _value;
    }

    void reset()
    {
        FastMutex::ScopedLock lock(_mutex);
        _value = 0;
    }

    void setInterval(long milliseconds)
    {
        _timer.setPeriodicInterval(milliseconds);
    }

    long getInterval() const
    {
        return _timer.getPeriodicInterval();
    }

protected:
    void onTimer(Timer& timer)
    {
        FastMutex::ScopedLock lock(_mutex);
        ++_value;
    }

private:
    long _value;
    Timer _timer;
    mutable FastMutex _mutex;
};

class CounterRequestHandler: public HTTPRequestHandler
{
public:
    CounterRequestHandler(Counter& counter):
        _counter(counter)
    {
    }

    void handleRequest(HTTPServerRequest& request,
                       HTTPServerResponse& response)
    {
        HTMLForm form(request, request.stream());

        if (form.has("interval"))
        {
            long interval = NumberParser::parse(
                form.get("interval"));
            _counter.setInterval(interval);
        }
        else if (form.has("action"))
        {
            std::string action = form.get("action");
```

```
            if (action == "reset")
                _counter.reset();
        }

        response.setChunkedTransferEncoding(true);
        response.setContentType("text/html");

        std::ostream& ostr = response.send();
        ostr << "<html><head><title>Counter</title>"
             << "<body><h1>Counter</h1>"
             << "<table>"
             << "<tr><th>Value:</th>"
             << "<form method='post'>"
             << "<td>" << _counter.value() << "</td><td>"
             << "<input type='hidden' name='action' value='reset'>"
             << "<input type='submit' value='Reset'>"
             << "</td></form></tr>"
             << "<tr><th>Interval:</th>"
             << "<form method='post'>"
             << "<td><input type='text' name='interval' value='"
             << _counter.getInterval() << "'>ms</td><td>"
             << "<input type='submit' value='Set'></td></form></tr>"
             << "</table></body></html>";
    }

private:
    Counter& _counter;
};

class CounterRequestHandlerFactory: public HTTPRequestHandlerFactory
{
public:
    CounterRequestHandlerFactory(Counter& counter):
        _counter(counter)
    {
    }

    HTTPRequestHandler* createRequestHandler(const HTTPServerRequest&)
    {
        return new CounterRequestHandler(_counter);
    }

private:
    Counter& _counter;
};


int main(int argc, char** argv)
{
    Counter counter;
    ServerSocket svs(8080);
    HTTPServer srv(
        new CounterRequestHandlerFactory(counter),
        svs,
        new HTTPServerParams);
    srv.start();
    Thread::sleep(100000);
    srv.stop();

    return 0;
}
```

*Beispiel 9: Applikation mit Web Benutzerschnittstelle*

Das Programm enthält zunächst eine C++ Klasse `Counter`, welche den Zähler implementiert. Der eigentlich interessante Code findet sich in der `handleRequest` Methode der Klasse `CounterRequestHandler`. Zunächst werden Formular-Daten ausgewertet. Je nachdem, ob ein neuer Intervall-Wert gesetzt wurde, oder der Reset-Button gedrückt wurde, enthalten die Formulardaten entweder ein Feld namens `interval` mit dem eingegebenen Intervall, oder ein Feld namens `action` mit dem Wert `reset`. Der Intervall-Wert wird als String geliefert, daher muss er noch in eine Zahl konvertiert werden, bevor er an das Counter Objekt übergeben wird. Schließlich wird das HTML Dokument generiert und an den Client gesendet. Der Rest des Programmes ist bereits aus Beispiel 7 bekannt. Abbildung 3 zeigt die fertige Benutzerschnittstellle der Applikation im Browser.

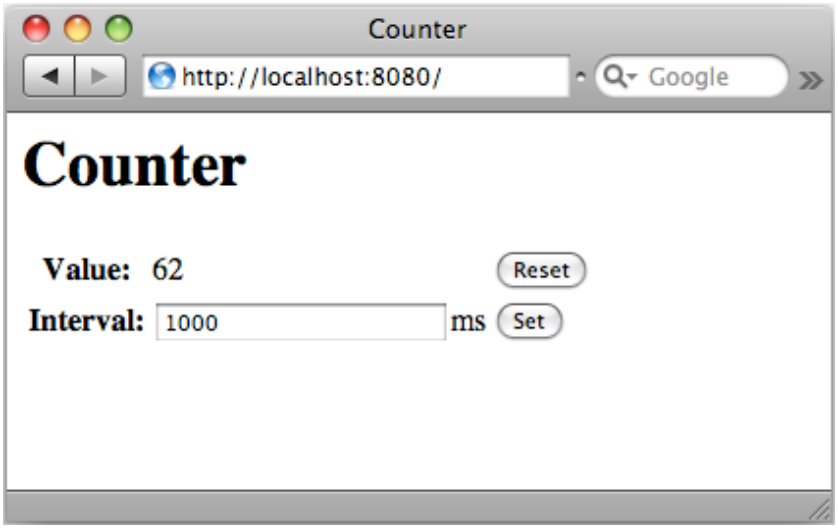


*Abbildung 3: Einfache Browser-basierte Benutzerschnittstelle*

## 6 Zusammenfassung und Ausblick

Der vorliegende Text hat gezeigt, wie eine Applikation mit einem integrierten Web Server versehen werden kann, um damit eine Web Browser-basierte Benutzerschnittstelle zu implementieren. Ist eine geeignete HTTP Server Bibliothek, wie z. B. libmicrohttpd oder die Net Bibliothek der POCO C++ Libraries verfügbar, reichen Grundkenntnisse des HTTP Protokolls, sowie HTML Kenntnisse aus, um eine Benutzerschnittstelle zu implementieren.

Seit einiger Zeit aktuell sind sogenannte Ajax[10]-basierte Benutzerschnittstellen. Der Grundgedanke hierbei ist, Javascript und asynchrone HTTP Anfragen einzusetzen, um den Inhalt einer im Browser dargestellten HTML Seite dynamisch zu verändern. Mit Hilfe von Ajax können sehr aufwändige Benutzerschnittstellen programmiert werden, die sich fast nicht mehr von herkömmlichen Benutzerschnittstellen unterscheiden. Dafür erforderlich sind jedoch tiefgehende JavaScript Kenntnisse. Es gibt bereits eine Vielzahl von JavaScript Bibliotheken, welche die Implementierung von Ajax-basierten Benutzerschnittstellen erleichtern sollen. Jedoch auch auf der Server-Seite ist eine Bibliothek zur automatischen Generierung von HTML und JavaScript Code für Ajax-Applikation sehr nützlich. Als Teil der POCO C++ Libraries ist z. B. die WebWidgets Bibliothek frei verfügbar.

[10] Ajax ist eine Abkürzung für *Asynchronous JavaScript and XML*