HTTP PROTOKOL PRO VÝUKOVOU HW/SW PLATFORMU FITKIT

HTTP PROTOCOL FOR TEACHING HW/SW PLATFORM FITKIT

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE
AUTHOR

ISTVÁN JÓBA

VEDOUCÍ PRÁCE
SUPERVISOR

Dr. Ing. OTTO FUČÍK

BRNO 2009

Abstrakt

Cílem bakalářské práce je implementovat protokol HTTP pro výukovou platformu FITkit. Po představení FITkitu a jeho částí se práce zaměřuje na implementační detaily protokolu HTTP, jako jsou jeho verze, základy komunikace, stavové hlášky a ověřování klientů. Implementace je založena na API knihovnách libfitkit a libkitclient, které byli vytvořeny pro FITkit.

Abstract

The goal of the bachelor thesis is the implementation of the HTTP protocol for the FITkit teaching platform. After introducing the FITkit and its components, the work deals with the Hypertext Transfer Protocol (HTTP), such as the versioning, communication concepts, status signaling. The implementation is based on the libfitkit and libkitclient APIs, developed by the FITkit team.

Klíčová slova

FITkit, HTTP, mikrokontrolér, síť, komunikace, libfitkit, libkitclient

Keywords

FITkit, HTTP, microcontroller, network, communication, libfitkit, libkitclient

Citace

István Jóba: HTTP protocol for Teaching HW/SW Platform FITKit, bakalářská práce, Brno, FIT VUT v Brně, 2009

HTTP protokol pro výukovou HW/SW platformu FITKit

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Dr. Ing. Otto Fučíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
István Jóba
27. 05. 2009

Poděkování

Na tomto místě bych chtěl poděkovat mému vedoucímu Dr. Ing. Ottovi Fučíkovi za odborné vedení, za konzultace a připomínek, které mi pomohly při řešení bakalářské práce.

Contents

1 Introduction

In the last one and a half decade the World Wide Web (WWW) and every technology inspired by it, like the HTTP protocol, experienced an exponential growth both in numbers of users and devices/platform supporting or using it. With the general technological evolution, the size of these devices is constantly shrinking, allowing deploying smaller and smaller WWW compliant devices, like the FITkit.

The FITkit project, a training platform to study hardware/software co-design, created at our faculty, allows the students to gain competitive skills in the field of embedded devices. In the second chapter I describe this platform: its main components, what devices/peripherals are available for it, the internal and external communication interfaces, and finally the software/firmware tools that aid the potential developer for the platform.

The next chapter explains the history and purpose of the HTTP protocol, describes the basic concepts of HTTP, the communication principles, and the main differences between protocol versions. Also deals with the design of a simple HTTP/1.0 compliant server developed for the FITkit, explaining the HTTP protocol in detail: message syntax, method usage, transferring entities, status messages, noting possible design changes and improvements in a further revision on the way.

In the last chapter we will see to the implementation details: including details on the reduced HTTP protocol in the application, the supported methods, the built-in sample webpage and its resources. Also describes the protocol created to facilitate communication between the FITkit and the PC. Detailed explanation is given about the structure of the two applications: the HTTP server developed for the FITkit and the relay application running on a PC, transforming TCP/IP communication back and forth to the HTTP server. This chapter also includes the description of the I/O interface which was created to allow the current, serial protocol based protocol to be easily replaced by a future TCP/IP stack.

2 FITkit

Microcontrollers and embedded systems are all around us: from a simple wristwatch to a camera or DVD player. All these devices have one thing in common: their hardware had to be designed and their software had to be developed by an engineer.

The goal of introducing the FITkit platform was to give the students at our faculty the opportunity to develop and design practical projects involving not only software but also hardware-based applications. In this chapter we will get familiar with this HW/SW platform.

2.1 Overview

Embedded systems are a successful and flourishing branch of IT these days, with great future potential: its products are used in everyday life in great quantity.

A typical embedded system (mp3 player, TV set, etc.) contains a microprocessor, specialized hardware and application software. This means the developer needs to be familiar with both software and hardware designing and how to use this knowledge in practice.

The FITkit teaching platform provides not simply a hardware background, but also a complete set of software tools and programming APIs for aiding the beginners and thus providing a great deal of practical knowledge and skill that a versatile embedded system engineer needs to have to be successful and competitive at the global labor market.

2.2 Components

The FITkit teaching platform can be divided into three main areas: hardware, firmware, software.

2.2.1 Hardware

First let’s examine the electrical components.

2.2.1.1 Microcontroller

The FITkit contains a microcontroller unit (MCU) from the MSP430 family manufactured by Texas Instruments. In the recent 2.0 revision of the FITkit the selected chip is the MSP430F2617T-rev E. It is built around a 16 bit RISC microprocessor with a large amount of peripherals and has very low power consumption, which makes it a great candidate for low powered and wireless portable applications.

The MCU key features are the following [1] :

- Low supply voltage, ranging from 1,8 V to 3,6 V
- Ultra low power consumption
  - - Active mode: 365 µA @ 1MHz, 2.2V
  - - Standby mode (VLO): 0.5 µA
  - - Off mode (RAM retention): 0.1 µA
- 16-Bit RISC architecture, 62,5-ns instruction cycle time
- 92KB+256B Flash memory, 8KB RAM (upgraded in FITkit rev. 2.0)

- Max. processor frequency 16 MHz @ 3,3 V
- Three-channel internal DMA
- Two 16 bit timers (Timer_A with three capture/compare registers and Timer_B with seven capture/compare-with-shadow registers)
- On-Chip comparator
- Four Universal Serial Communication Interfaces (USCI) supporting I^2C, synchronous SPI, IrDA, enhanced UART with auto-baud rate detection
- Bootstrap loader,
- 48 I/O ports
- Temperature sensor

The functional block diagram of the MCU is shown on Fig. 1:

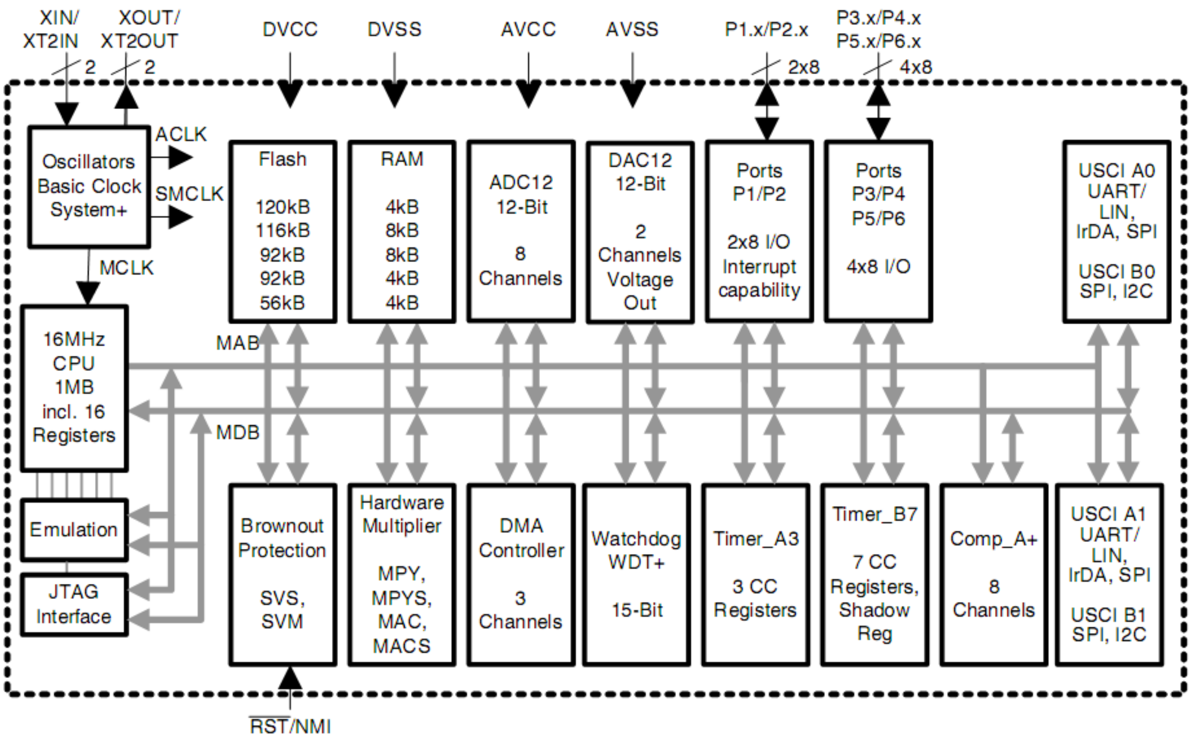

XIN/ XT2IN
XOUT/ XT2OUT
2
2
DVCC
DVSS
AVCC
AVSS
P1.x/P2.x
2x8
P3.x/P4.x P5.x/P6.x
4x8
ACLK
SMCLK
MCLK
Oscillators Basic Clock System+
Flash 120kB 116kB 92kB 92kB 56kB
RAM 4kB 8kB 8kB 4kB 4kB
ADC12 12-Bit 8 Channels
DAC12 12-Bit 2 Channels Voltage Out
Ports P1/P2 2x8 I/O Interrupt capability
Ports P3/P4 P5/P6 4x8 I/O
USCI A0 UART/ LIN, IrDA, SPI
USCI B0 SPI, I2C
16MHz CPU 1MB incl. 16 Registers
MAB
MDB
Emulation
JTAG Interface
Brownout Protection SVS, SVM
Hardware Multiplier MPY, MPYS, MAC, MACS
DMA Controller 3 Channels
Watchdog WDT+ 15-Bit
Timer_A3 3 CC Registers
Timer_B7 7 CC Registers, Shadow Reg
Comp_A+ 8 Channels
USCI A1 UART/ LIN, IrDA, SPI
USCI B1 SPI, I2C
RST/NMI

Fig. 1: Functional block diagram of MSP430F2617, [1] p.9

2.2.1.2 FPGA module

The next main component of the kit is the FPGA chip (Field Programmable Gate Array) containing many thousands of logic gates: their configuration can be reprogrammed (rewired) using a hardware description language or a schematic design.

It’s able of performing any logical function that an application-specific integrated circuit can; with the added ability of easily redesign and redeploy a new or improved application. The use of a programming language for hardware design makes the process of designing hardware components more available and easy for the students. While the realized implementation doesn’t deal with this component directly, it does use it with the help of the libfitkit library and other components and further version also can make use of it more intensively.

The FITkit contains a XC3S50-4PQ208C chip from the Spartan3 family manufactured by Xilinx, with the following characteristics [2] [3]:

- 192 configurable logic blocks in 16 rows and 12 columns, 1728 logic cells, 50k system gates
- 72 Kbits of block RAM and 12 Kbits of distributed RAM
- Four dedicated 18 bit multipliers
- Up to 124 user I/O ports, support for 23 I/O standards

2.2.1.3 USB to serial/parallel communication interface

For communicating over USB the FITkit utilizes the FT2232D chip from FTDI. It offers the kit two independent communication channels (A and B).

Channel A is connected to the Spartan 3 FPGA and enables a PC application to communicate via any device created in the FPGA via protocols like I^2C or connect channel A to the serial channel and communicate with the device via the virtual COM port in the computer.

Channel B is connected to the microcontrollers programming pins (RESET, TST) and to MCU UART’s (Universal asynchronous receiver/transmitter) RxD and TxD (receive and transfer) pins. When using the ***libfitkit*** API created by the FITkit developers, this channel is used for communication between the FITkit application and the terminal, and also for programming/flashing new application to the FITkit [2].

2.2.1.4 Other components

- LCD display (16 characters each in two lines)
- 4x4 keypad
- Audio interface (in/out)
- Two PS/2 connectors
- VGA interface
- RS232 connector
- DRAM 8x8 Mbit
- I/O connectors for external peripherals
- Flash memory for the FPGA application code

The layout of the components can be seen on Fig. 2: with the keypad and LCD display dismounted on the right-hand picture to reveal the FPGA and DRAM module beneath them:

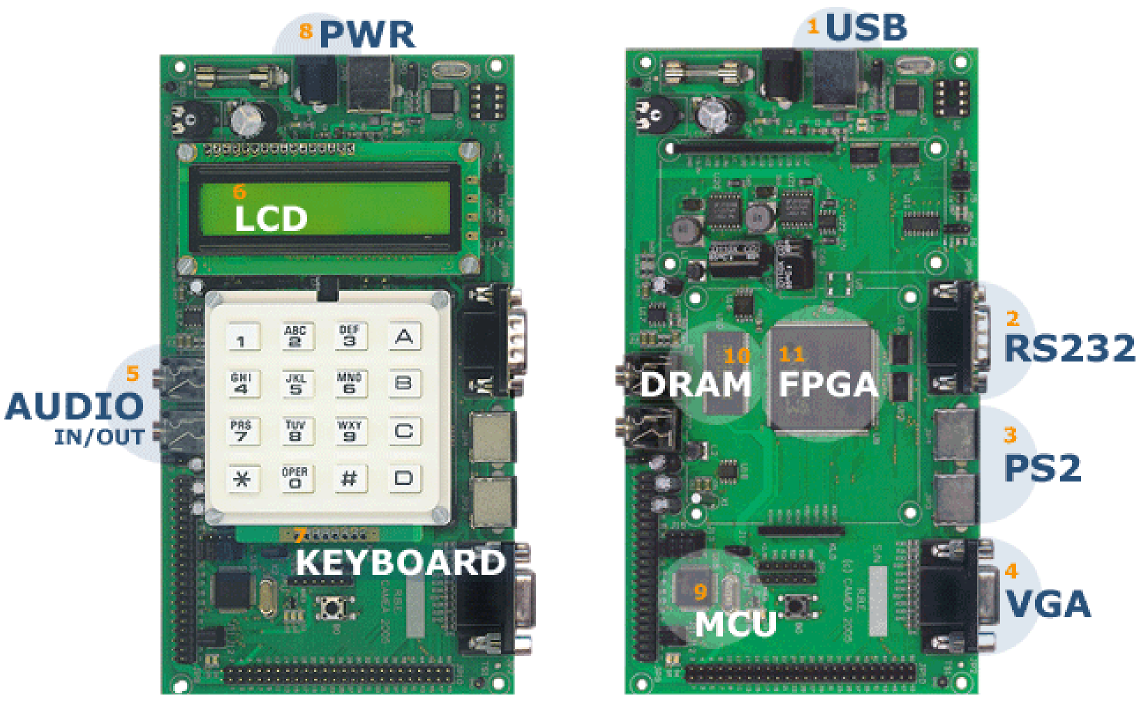


Fig. 2: FITkit components [2]

2.2.2 Firmware

The firmware of the FITkit deals with the following issues.

2.2.2.1 Communication between the main components

Almost every component and peripheral needs to be connected to the microcontroller in some way or other, so the embedded application can use it. While some of them (like the USB interface) are directly connected, most of the built-in peripherals are connected to the FPGA chip.

The communication between the MCU, FPGA and the Flash memory is realized using the SPI interface (Synchronous/Serial Peripheral Interface). It is a high speed full-duplex synchronous master/slave serial bus using four wires controlled by the master device:

- CS (chip select): indicates the start of a data frame, generated by the master and active at low
- SCK (clock): generated by the master
- MOSI (Master Out Slave In): data output for master, input for slave
- MISO (Master In Slave Out): data input for master, output for slave

This interface is used when the MCU application needs to communicate with FPGA devices, typically the memory controller or a device controller.

In the FITkit the MCU, FPGA and Flash memory are using the same SPI signals (Fig. 3:) so the slave device is selected by the first byte of transferred data. If the first byte indicates that the communication is intended for the FPGA the address of the specific device is transmitted by the MCU.

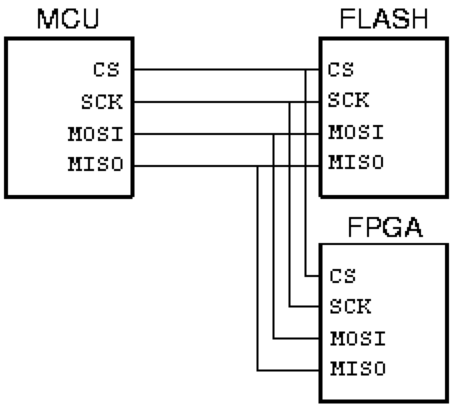


Fig. 3: SPI used internally on FITkit [2]

2.2.2.2 Device controllers

The next step is to create the device controllers for the peripherals included in the FITkit, so they can be re-used in later applications designs. The controllers are described in VHDL, a hardware description language, and are communicating with the microcontroller via the SPI interface described earlier.

The following peripheral controllers already have been implemented:

- LCD display controller, a generic interrupt controller, 4x4 keypad, PS/2 controller and SDRAM controller.

2.2.2.3 Library libfitkit Device “drivers”

This library contains modules for reprogramming the kit, offers communication via a command based terminal which can be used in user applications for PC communication, and also offers C macros to simplify development (string compare, character and string conversions, etc.). It was developed in C.

Many device “drivers” were added to this library. This allows to easily use the implemented device controllers (residing in the FPGA) in an application (running on the MCU)

The library contains functions such as:

- Communication interface between FPGA devices (the controllers) and the drivers (MCU) via the “FITkit SPI”.
- A simple terminal application for communication via USB and UART, including a command interpreter for controlling the MCU
- FPGA configuration: a basic task of the library is to load the FPGA configuration data after initialization (e.g. power on or reset).

2.2.3 Software

After identifying the hardware and firmware components of the kit, we can move to the development aids available for the PC.

2.2.3.1 Library libkitclient

The libkitclient is a multiplatform library for managing and interaction with FITkits. It was written in C++ (with binding for the Python language) and offers a uniform API for Windows and Linux. It is using the FTDI library to directly access the USB controller on the FITkit which simplifies the application configuration to exclude any hassle with configuring virtual COM ports.

Brief list of operations offered in the library:

- managing multiple connected FITkits
- change the communication channel properties (speed, parity, stop bits, etc.)
- reset MCU
- write or read data

QDevKit

An example of using the libkitclient: it's a multiplatform application with GUI, created to simplify the work with– and in some degree the development for – the FITkit. Its main features are:

- discovers connected FITkits and allows communicating with them via a terminal (it is using the libkitclient API for these tasks)
- acts as a front-end for compiling and flashing both MCU and FPGA based FITkit applications (projects), while maintaining a fresh copy of the applications via SVN (Subversion: a version control system)
- allows creating plug-ins

3 Hypertext Transfer Protocol (HTTP)

This chapter explains the history and purpose of HTTP, describes the basic concepts of HTTP, the main differences between protocol versions. In the second part explains the structure of the HTTP message more deeply and giving notes about implementing the protocol in an embedded device.

3.1 Overview

The **Hypertext Transfer Protocol** (HTTP) was created at the European Organization for Nuclear Research (CERN) as part of the **World Wide Web** (WWW) project. The project's main goal was to assist the sharing of information among researchers using hypertext documents. Digitally linking related documents and by selecting the reference would cause the linked document to be retrieved.

3.1.1 Basic concepts

HTTP is a stateless request/response protocol which operates at the application layer of the OSI Reference Model and the Internet Protocol Suite (commonly known as TCP/IP). While it is mostly used in TCP/IP networks the HTTP protocol itself requires only an error-free two-way communication and can be implemented on top of any network with reliable transport layer. Over its 19 years of existence the protocol has seen three greater revisions (versions 0.9, 1.0 and 1.1) and numerous features were added while the basic concepts remained the same.

The basic concepts are [4], [5]:

3.1.1.1 Message

The *HTTP message* is the basic unit of HTTP communication. It consists of either a *request* or a *response*. The *message* needs to be transmitted via an error-free protocol (most of the time this is TCP).

Its syntax has changed over time so new features could be implemented. In the latest versions the message format is similar to that used by e-mail (Interne Mail) and the MIME standards (Multipurpose Internet Mail Extensions).

3.1.1.2 Request

At the beginning of the HTTP communication a *request* is sent by the *user agent* (or *client*) to the *server*. In the first line of the message it contains the *method* to be applied to the *resource*, the identifier of the resource, and starting with HTTP version 1.0, the protocol version too. Depending on the method and the protocol version it can also contain additional headers and an *entity*.

3.1.1.3 Response

An *HTTP response* is sent as an answer for a *request*. In HTTP/1.0 and 1.1 it contains a status line informing the client about the result of the request. Depending on the interpretation of the request it may also contain additional headers and an *entity*. (see HTTP/0.9 description for the old response type)

3.1.1.4 Entity

An *entity* is a representation of a *resource*. It consists of metainformation in the form of the *entity header* and the content in the form of *entity body*. Both HTTP *request* and *response* may contain an entity (header only or whole entity).

3.1.1.5 Resource

A *resource* is a network data object or service which can be identified by an URI (Uniform Resource Identifier).
As far as HTTP is concerned, URIs are simply formatted strings which identify via name (in the form of Uniform Resource Names – URN), location (in the form of Uniform Resource Locators – URN), or any other characteristic of a network resource.

3.1.1.6 Client or user agent

The *client* is a program that establishes a connection to the *server* for the purpose of sending *requests*. These are often browsers, spiders (web-traversing robots), or other end user tools. As of HTTP/1.1 the clients should recognize the status-line of every HTTP 1.0 or 1.1 response and understand any valid response in the format of HTTP 0.9, 1.0 or 1.1.

3.1.1.7 Server or origin server

An application that accepts connections in order to service *requests* by sending back *responses*. For backwards compatibility the servers must recognize the request line of previous protocol versions, understand any valid request in previous formats and finally respond accordingly to the major version of the client protocol.

3.1.1.8 Proxy

It is an intermediary program which acts as both a *server* and a *client* for the purpose of making requests on behalf of other clients. A proxy must interpret, and, if necessary, rewrite a request message before forwarding it. Proxies are often used to implement content filtering and save bandwidth by caching previous responses and resources.

3.1.1.9 Gateway

It is a server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway. Gateways are often used as server-side portals through network firewalls and as protocol translators for access to resources stored on non-HTTP systems.

3.1.1.10 Communication examples

To summarize the basic concepts of HTTP let’s see some examples.

Our first example while being the simplest possible scenario, it is also the most common one: with an HTTP client sending a request through the communication channel and than the server sending back a response (Fig. 4:).

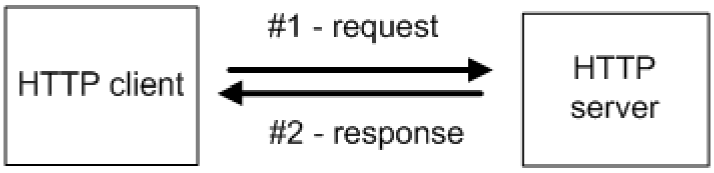


Fig. 4: HTTP request/response in a simple client/server environment

The next example (Fig. 5:) introduces a forwarding agent (a proxy) to the communication chain. Upon receiving a client request, the proxy needs to analyze it and act correspondingly with its internal rules (e.g.: changing the request, providing the requested resource from its internal cache, denies forwarding the request, provide content filtering, etc.).

Utilizing a proxy is common in corporate environments where it is used mainly for its content filtering options.

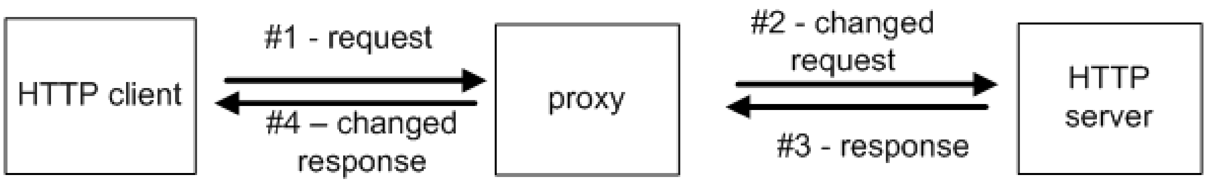


Fig. 5: Adding a proxy to the HTTP communication chain

3.1.2 Protocol versions

HTTP uses a "<major>.<minor>" numbering scheme to indicate versions of the protocol. The <major> number is incremented when the format of a message within the protocol is changed while the <minor> number is incremented when the changes made to the protocol add features which do not change the general message parsing algorithm, but which may add to the message semantics and imply additional capabilities of the sender.

Including the HTTP-Version field in the request and response messages (part of the standard from v1.0) is intended to allow the sender to indicate the format of the message and its capacity for understanding further HTTP communication. If the version field is missing the receiving application must assume the message is in the simple HTTP/0.9 format.

The subsequent versions were created with backwards compatibility in mind and to ease the implementation of compatible clients, servers and proxies alike, and promoting the robustness of the applications using HTTP as described in *Use and Interpretation of HTTP Version Numbers* [6].

3.1.2.1 HTTP / 0.9, or the protocol as defined in 1991

The first version of HTTP (HTTP/0.9) was a very simple protocol to promote the quick adaptation and easy implementation of the WWW. It was first published in 1991 by its author Tim Berners-Lee [7].

The definition of this protocol used the TCP protocol as the preferred transport layer to explain the client/server communication, although at the same time permitting implementations with other connection-oriented services [7]. Later versions also had strong connections with the TCP/IP protocol suite, including built-in optimizations for TCP/IP networks.

The downside of being a very simple protocol it lacked most of the functionalities of a modern interactive protocol the evolving WWW needed: supporting only GET requests (meaning one-way information transfer, from server to client), and the transfer of hypertext-only documents in those requests.

The format of the GET request method (recent versions referring to this as the “Simple-Request”) was the following:

```
Simple-Request = "GET" SP Request-URI CRLF              [4]
```

A server would respond with a byte stream of ASCII characters containing the HTML document specified by the `Request-URI` and close the connection.

It must be noted that the shortcomings of the protocol were soon discovered and a year later many of them were corrected or improved.

3.1.2.2 HTTP / 1.0

The first formalized version of the protocol was version 1.0, described in RFC1947 and published in May 1996 [4].

This version was a great improvement which added numerous new features to the protocol. The most important addition was the generalization of the protocol: incorporated MIME-like concepts and header structure and added support for other types of data beside hypertext.

While the new features allowed creating more sophisticated web servers and clients, it was still backward compatible with older clients and servers using HTTP/0.9.

New features in this version (among others):

- Including the HTTP-Version field in the first line of the message in every request and response (for backwards and further version compatibility).
- Defining new request methods beside GET: HEAD and POST, while leaving the list open-ended for extensions.
- Status codes in response messages to indicate the result of the previous request
- MIME-like message format, proxy support, etc.

The high success and rapid development of the Web soon revealed some of the HTTP/1.0's design limitations, namely:

- Only one website could be hosted with one server/port combination (IP or domain name + TCP port). Workarounds: additional servers for every website (not very economic), or each website listening to another port (not very user-friendly)
- For every HTTP request a new TCP connection needed to be established, with it's three-way opening handshake (and possibly a four-way handshake closing the connection) and thus increasing the delay of retrieving resources from the server.

3.1.2.3 HTTP / 1.1

The shortcomings of HTTP/1.0 mentioned before were remedied in the next version. HTTP/1.1 was made available in 1999 as RFC 2616 [1] and introduced several significant improvements over version 1.0 of the protocol.
The issues pointed out in the previous section were also addressed:

- Multi-homed web servers: every 1.1 request now MUST contain the new Host header. This contains the hostname and port of the requested resource as given by the user or the referring resource. Additionally the new standard for HTTP/1.1 states that all implementations of HTTP (including updates to existing HTTP/1.0 applications) MUST support this header.
- Persistent connections: a HTTP/1.1 client may send multiple requests in one TCP connection. This saves CPU time in network routers and hosts as well, reducing network congestion and allowing time for the protocol to determine the congestion state of the network, latency is improved because no subsequent TCP opening handshakes are needed. The RFC doesn't define this feature as mandatory for a valid HTTP/1.1 implementation, only as a "should be implemented".

3.2 Analysis

In the previous chapter we got familiar with the basic concepts of the HTTP communication and the differences between its versions.

The main differences between versions 1.1 and 1.0 are in the variety of supported header fields (newer has more) and most of them are not mandatory, and thus the features of **HTTP/1.0** I am considering to be enough for a FITkit HTTP application. Even more the hardware limitation in form of relatively small RAM onboard the microcontroller makes implementing the whole standard more complicated.

Considering the fact that out of the three protocol versions only HTTP/1.0 and HTTP/1.1 was formed as a standard in an RFC document, and any recent application would implement one of these versions, we will not mention RFC notes about HTTP/0.9 compatibility issues.

Any implementation-specific notes are *highlighted*, most of them are regarding memory limitations for the implementation of the HTTP server, alternatively a client.

3.2.1 HTTP Message

As we discussed earlier the basic communication unit of HTTP is the *message*. The shared major version number of the 1.1 and 1.0 standards guarantees that the message format will be similar.

Both HTTP versions define two types, request and response message:

```
HTTP-message  =  Request | Response                     [5]
```

The request and response message follow the same *generic message* format, consisting of a start-line, zero or more message headers (all ended by exactly one CRLF), an empty line indicating the end of the message headers, and finally an optional message body (entity body):

```
generic-message  =  start-line
                    *(message-header CRLF)
                    CRLF
                    [ message-body ]                     [5]
```

The *start-line* is of course defined separately for the request and response message:

```
        start-line  =  Request-Line | Status-Line       [5]
```

3.2.1.1 Message headers

Many differences between HTTP 1.0 and 1.1 are in these headers: the newer version defines more.

The message headers include the *general*, *request*, *response* and *entity* headers. Each *header field* consists of a *header name* followed by a colon (':') and the *field value*.

The order of the headers is not significant although it is a good practice to send them in the following order: general headers, request or response headers, and finally entity-headers.

Multiple message headers with the same field header might be included in a message only if the field values can be combined into one value separated by a comma without changing its semantics. Proxies are forbidden to change the order of these headers.

General header fields

General headers fields are applicable for both request and response messages, excluding any entity transferred within them.

The two general header fields in HTTP/1.0 are *Pragma* and *Date*. The former is being used in the request message by a client to bypass caching in the HTTP communication chain. The latter represents the date at which the message was originated and can be used by both origin servers and user agents while user agents should only include it when transferring an entity (like in the POST method).

<u>*Implementation note*</u>*:* None of these headers are mandatory, but including the Date header in the implemented HTTP server might be a worthy addition in a future version. It involves creating a date/time module with appropriate external synchronization..

3.2.1.2 Message/entity body

The entity body (if any) sent with an HTTP request or response is in a format and encoding defined by the entity header fields.

A request message contains an entity body only if the method allows it. The presence of a message body is signaled by the *Content-Length* header field, which is mandatory for every HTTP/1.0 request containing a body.

In the response message the presence of a body is dependent on both the request method and the response time. A response to the HEAD method must not contain a body, although the Content-Length header field is mandatory. Also some status messages are forbidden to have bodies: all 1xx (informational), 204 (no content), 304 (not modified).

Message type

Every message containing an entity body should include the *Content-Type* and *Content-Encoding* header fields. The former is indicating the data type and the latter a possible coding of the data (usually meaning compressing).

Implementation note: compressing the entity body doesn't make sense for low amounts of data, like that the FITkit is able to process at one time.

3.2.1.3 Message length

Implementation note: The message length can be deduced for every type of message in reasonable limits. This is extremely important for determining right buffer sizes on the FITkit.

The standard requires every GET and HEAD request line and their header fields to end with exactly one CRLF and ending the whole message with two CRLFs, so most of the request messages can be read line after line.

If the request message includes an entity body (with arbitrary number of CRLFs), a *Content-Length* header field MUST indicate its length. At this point it can be decided whether the FITkit application can handle the message or discard it.

3.2.1.4 Request

The first line of the request message contains the *method* to be applied to the *resource,* the identifier of the resource and the protocol version (the latter being "HTTP/1.0" for version 1.0).

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF      [4]
```

Example: GET /index.html HTTP/1.0 <CRLF>

Now let's see what request header fields and methods (commands) are usually offered by HTTP/1.0 implementations (the method token is case-sensitive, unlike most of the standard).

3.2.1.5 Request header fields

The request header fields allow the client to pass additional information about the request, and about the client itself, to the server.

It can contain the following header fields:

- *Authorization*: this field is used after receiving a response with 401 status line and if the user agent wants to authenticate with the server. The user and password are encoded in base 64. Example:

  Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ== [4]
- *From*: should contain an Internet e-mail address for the human user who controls the requesting user agent. Rarely used

- *If-Modified-Since*: used in conjunction with the GET method (see below)
- *Referer* [sic]: in this field the client may specify the address where the Resource-URI was obtained.
- *User-Agent*: contains information about the user agent, for the Opera browser version 9.25 running on Windows XP it looks like this: Opera/9.25 (Windows NT 5.1; U; en)
- *Host*: it was specified in version 1.1 and noted that every new or upgraded 1.0 application MUST include it. It is used for conserving IP addresses by enabling multiple websites to be hosted on the same port and same server. Its content is the host and port from the original URI.

3.2.1.6 GET

The GET is the most common HTTP method. It means to retrieve whatever information (in the form of an entity) specified by the Request-URI.

It can be modified to a “conditional GET” with the If-Modified-Since header field: meaning the resource will only be transferred if it is newer than the specified date/time. This is intended to reduce network usage without transferring unnecessary data.

Implementation note: although the HTTP server running on FITkit is only HTTP/1.0 compliant, it is bound to support requests with the same major version number, e.g. a web browser. The modern browser is using the more customizable HTTP/1.1 to deliver a user-friendly, localized web-experience to the average user, and this mean that on the average a GET request is around 300-500 bytes. As we will see from the example, it contains much more information about the requesting client’s capabilities.

Example: Opera/9.25 accessing the website at www.cdr.cz

```
GET / HTTP/1.1
<CRLF>
User-Agent: Opera/9.25 (Windows NT 5.1; U; en)
<CRLF>
Host: www.cdr.cz
<CRLF>
Accept: text/html, application/xml;q=0.9,
application/xhtml+xml, image/png, image/jpeg, image/gif,
image/x-xbitmap, */*;q=0.1
<CRLF>
Accept-Language: hu,cs-CZ;q=0.9,cs;q=0.8,sk;q=0.7,en;q=0.6
<CRLF>
Accept-Charset: iso-8859-1, utf-8, utf-16, *;q=0.1
<CRLF>
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
<CRLF>
Connection: Keep-Alive, TE
<CRLF>
TE: deflate, gzip, chunked, identity, trailer
<CRLF>
<CRLF>
```

Our server would discard most of the header-fields, for either being 1.1-only header fields (like TE, Connection), or not useful for a simple HTTP server: e.g. the User-Agent is used for creating visitor statistics, and it is very unlikely that the same resource would be available in different languages or character set due to space restrictions (although on-the-fly converting might be achievable).

3.2.1.7 HEAD

It is identical to GET except the response must not contain ANY message-body. This method can be used for obtaining metainformation about the resource, because the server should generate the same message for an identical GET or HEAD request (save for the message body). This method is often used for testing hypertext links for validity, accessibility (response message's status line) and recent modification (Date field). It is worth noting that this method, in contrast to the GET method, doesn't support the If-Modified-Since header-field.

Implementation note: this method is great for a HTTP client with limited memory available and for a server application shouldn't be too difficult to implement. While usually a request's size is in the same range (because most of them are either GET or HEAD), a web client parsing an unfamiliar HTML document must pay attention as larger message bodies are more likely to be encountered. A client using this method (and a supporting server) could deal with such inconvenient situation at the application level in contrast with the more extreme TCP connection dropping or ignoring the incoming data at the expense of wasting bandwidth (not necessarily acceptable for every application/platform).

3.2.1.8 POST

The POST method is used to request that the destination server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. POST is designed to allow a uniform method to cover the following functions:

- Annotation of existing resources
- Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles
- Providing a block of data, such as the result of submitting a form to a data-handling process
- Extending a database through an append operation.

A successful POST does not require that the entity be created as a resource on the origin server or made accessible for future reference. That is, the action performed by the POST method might not result in a resource that can be identified by a URI. [4]

Implementation note: implementing the POST method in a server depends on the *type* of this server. By far the most common action desired is the third option above: it is part of the Hypertext Markup Language (HTML, RFC1866/2854) standard and it's needed for a web server.

If the server application determines that the POST method is not applicable for the given resource, it should respond with a 501 Not Implemented status code.

3.2.1.9 Other methods

Although so far we were focusing on HTTP/1.0 properties, I think it is worthy to note that many other methods exist for HTTP/1.1 or as standalone extensions.

To give some overview, here are the additional **HTTP/1.1** methods:
OPTIONS, PUT, DELETE, TRACE, CONNECT [5]

Some of the **WebDAV** (or just DAV) extensions:
COPY, MOVE [8]

3.2.2 Response

The answer for a successfully interpreted client request consists of a status line, zero or more general/response/entity header and optionally an entity body:

```
Response-message = Status-Line
                   *( General-Header
                   | Response-Header
                   | Entity-Header
                   CRLF
                   [ Entity-Body ]            [4]
```

Implementation note: A response header is fairly less interesting then a request header, both for a client and a server, mainly for the less variable headers it can contain.

3.2.2.1 Response header fields

These fields are intended to allow the server to pass additional information about the server:

- *Location*: The Location response-header field defines the exact location of the resource that was identified by the Request-URI. For 3xx responses,the location must indicate the server's preferred URL for automatic redirection to the resource
- *Server*: it is used for sending more information about the server, like its name and version
- *WWW-Authenticate*: it must be included in 401 (unauthorized) response messages. Example: WWW-Authenticate: Basic realm="WallyWorld"

3.2.2.2 Status line and status codes

The first line of a response is always a status line and it looks like this:

```
Status-Line = HTTP-Version SP Status-Code
              SP Reason-Phrase CRLF        [4]
```

At the very beginning of the line the HTTP-Version token is the familiar “HTTP/1.0” literal.

The status-code is a three-digit number representing the result of the request interpretation and is meant to be machine-readable, that is processed by automata and is defined by the standard. In contrast, the reason-phrase is intended for humans and its contents are flexible, e.g. they can be translated to the server’s local language.

The most important and common status codes:

Informational 1xx

Not used, but reserved for future use. HTTP/1.0 does not define any 1xx status codes and they are not a valid response to a HTTP/1.0 request. (valid in HTTP/1.1 though)

Successful 2xx

This class of status code indicates that the client's request was successfully received, understood, and accepted.

- **200 OK**

The request has succeeded.

- **201 Created**

The request has succeeded and the new resource requested was created. From the HTTP/1.0 specification, only POST can create a resource.

- **204 No Content**

The request has succeeded, but there is no new information to send back.

Redirection 3xx

Further action must be taken in order to complete the request.

- **301 Moved Permanently**

The requested resource was moved to another known URL. This new URL is sent back in the Location field of the response, but the entity also includes its new location (usually in HTML).

- **302 Moved Temporarily**

The requested resource was temporarily moved. The old URL should be used in subsequent requestst (unlike in 301 Moved Permanently).

- **304 Not Modified**

Used if the client requested a resource with the If-Modified-Since field and the resource didn’t change since then

Client Error 4xx

These error codes are used when the requesting client seems to make a mistake. Except for a HEAD request, the server should send an entity also with the brief description of the issue

- **400 Bad Request**

The request's syntax was not valid and could not be understood by the server.

- **401 Unauthorized**

The request requires user authentication. Subsequent requests for this resource should include the Authenticate field. If the Authenticate field was already present it means that the provided credentials were refused.

- **403 Forbidden**

The server understood the request, but is refusing to fulfill it. Authorization will not help and the request should not be repeated.

- **404 Not Found**

The requested URI was not found on the server.

Server Error 5xx

These errors represent server-side failures.

- **500 Internal Server Error**

The server encountered an unexpected condition which prevented it from fulfilling the request.

- **501 Not Implemented**

The server does not support the functionality required to fulfill the request.

- **502 Bad Gateway**

The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.

- **503 Service Unavailable**

The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay.

4 Design and implementation

As we discussed, the HTTP protocol is a client/server protocol. This offers two choices for implementation: while a *client* could be implemented, the built-in memory arrangement of the kit is more suitable for a hosting (*server*) operation: in a HTTP client most of the application data would need to be obtained by the client during operation, needing greater amount of RAM to store it, while a server application can store its data in the built-in flash memory during programming.

Another argument for a server: the protocol itself is fairly universal. It offers transferring different/any types of data while enabling limited bidirectional communications. Creating a new custom application using these properties could be a possible way of implementing the protocol, the wide availability of *web browsers* offered the chance to have many available clients using a server running on the FITkit immediately after implementation: if this server happened to host a simple webpage.

This implementation of a HTTP server consists of two modules:

- the HTTP server (with two parts: the I/O interface and the server itself),
- relay application ("translating" between HTTP server and internet clients)

The HTTP server is developed for the microcontroller, while the relay application is running on a PC. The two devices are connected via the USB port present on the FITkit and are using a simple protocol for controlling behavior and transferring data.

4.1 The relay application

In this chapter I will describe how does the relaying application work and communicate with both the TCP/IP network and the HTTP server.

The relay application is implemented in C/C++. For the TCP/IP networking part the Winsock API was used, while the FITkit management is handled by the libfitkit API.

It consists of two source code files: `relay.cpp` and `relay.h`. When describing the behavior of the application I will also include which function is handling it, if applicable.

4.1.1 Designing the communication protocol

The HTTP protocols "default" network is TCP/IP. Unfortunately at the time of writing this thesis the FITkit platform wasn't equipped with a fully functional Ethernet layer component, nor offered some other way to communicate via TCP/IP.

However it did have other means of communication. It could communicate via its USB interface with a computer and a simple API was available for using it in the libfitkit library: the *uart* module.

The module implemented the serial I/O operations via small buffers to avoid (minor) data losses when other applications used the microcontroller. As an HTTP server potentially can receive and send substantially more data, filling up the buffers, I chose to workaround this issue within the application by defining which data transfer can possibly occur in each situation. Unfortunately this means that this particular implementation of the transfer protocol cannot be used with other applications relying on the UART module for communication.

Furthermore it is a highly application-specific protocol, basically operating on the *application level* of the ISO/OSI reference model, ignoring or simplifying some details of TCP/IP, so it is most probably not reusable in a different application either.

The goal of this protocol - was while having in mind the properties of the simple serial protocol - to create a communication protocol between the two parts of the application:

- The HTTP server which is running on the FITkit
- the relaying application listening for client connections on a TCP/IP network and sending it to HTTP server via the bridging protocol (running on PC)

While heavily using and depending on the libraries and components of the FITkit project, both applications implementing the final version of the protocol aren't meant to be integrated with the terminal functionality of the libfitkit: there were compatibility issues with using the shared communication channel and the attempts made to handle these issues proved to be error-prone.

The relaying applications behavior is controlled with sending control codes via Channel B by the HTTP server:

- communication via TCP/IP,
- transferring application specific data (HTTP request or respond),
- and controlling TCP states.

4.1.2 General initialization

The relay applications initialization consists of:

Parameter handling

The application has two possible parameters; the first is to display a simple help text and the second to specify the number of the TCP port where it will listen for client connections. Valid ports are: 80 and the range 1025-65500.

Discovering, managing FITkits

In the next stage the application uses functions from the *libkitclient* to find and initialize any FITkits connected to the computer.

In the final step Channel B of the USB-serial interface is opened to communicate with the microcontroller.

If any of the previous steps fail, the application will exit.

Network initialization

This consists of the Winsock initialization, registering the Winsock cleanup with standard library function `atexit()`, and opening the TCP port for listening in the `open_port()`function. The application is using a blocking socket, with one backlog connection.
The network handling part was mostly inspired by the Winsock tutorial available on MSDN [10].

FITkit initialization

After resetting the microcontroller, the relay application echoes initialization information sent by the FITkit (like information about FPGA programming and hardware initialization, sent by *libfitkit*).

4.1.3 Compiling

As mentioned, this module was developed in C++ using the Winsock API. For successful compiling the project needs the *libkitclient* header and library files to be in their respective directories under the `relay.c/.h` source file's directory structure. The included makefile contains every necessary argument (define, library and header file locations) and can be changed if necessary.

Development and testing was done using MinGW compiler system, on Windows XP and Windows Server 2008.

4.2 Communication protocol

The communication protocol is best to be described from the implementation point of view in the *relay application*.

The protocol itself is using control codes (opcodes) for signalization. These include (as defined in the `relay.h` header file):

- ACK (acknowledge: operation, data transfer, etc),
- NACK (operation failed),
- ACCEPT_CNN (accept connection, connection accepted),
- INCOMING_CNN (incoming connection),
- READ_X (read from…) and
- SEND_DATA (sending new data).

The opcodes are one byte long, and their interpretation are dependent on the current state of the application (no escape characters are used).

In this protocol the relaying application is essentially acting as a server, waiting for incoming opcodes from the application (the HTTP server) and acting according to them. Beside the initial handshake, the other commands (or chain of commands) can be received more than once, and are in fact processed in an infinite loop (the relay application cannot be terminated by the HTTP server with an opcode).

4.2.1 Initial handshake

This is a very simple handshake: the relay application waits for an initial ACK from the FITkit and upon receiving answers with another ACK.

This is the only read operation in the relay application with a timeout: if no communication attempt is made by the kit in the given time period (5 seconds), it is assumed that another application is running in the FITkit.

On the other hand, the kit also tries to perform syncing multiple times, so it is not necessary to perform the reset in the relay application during its initialization.

4.2.2 Accept a connection

If this control code (ACCEPT_CNN) is sent by the HTTP server, the relaying application first checks if any TCP connection is open. If there isn't, informs the HTTP server by sending an ACK that accepting a connection is now possible, and then proceeds accepting a TCP connection by calling the `accept_connection()` function. When a client makes a connection, the relay application sends the ACCEPT_CNN opcode to the kit (now meaning the connection was successfully accepted).

Waiting for a connection will not trigger a timeout.

On Fig. 6: we can see the successful accept connection command chain (note: between commands 2 and 3 a substantial delay can occur).

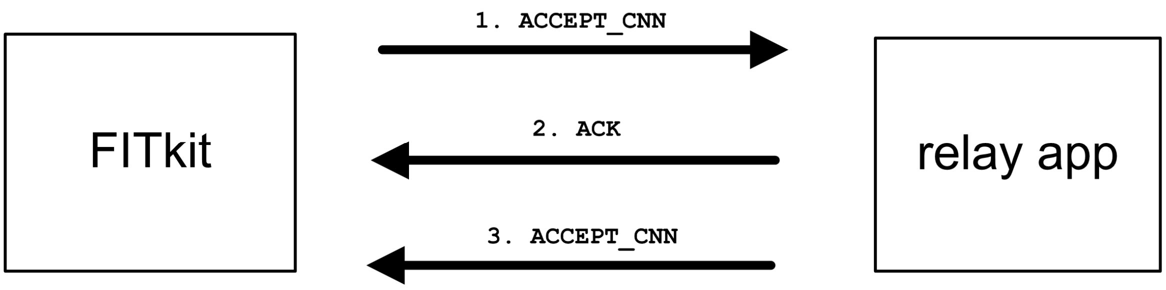


Fig. 6: A successful ACCEPT_CNN initiated by the application on FITkit

4.2.3 Close connection

Simply checks for an open connection, and if finds one, closes it. An ACK is sent back upon completion. Opcode: CLOSE_CNN.

4.2.4 Read request

This opcode (READ_X) instructs the relay application to read a request from an open connection. It is handled by the `send_data2fitkit()` function.

Upon successfully receiving data from the connected TCP client, the function sends the kit an ACK, immediately followed by a SEND_DATA opcode, to let it prepare for a data transfer. When ready, the kit signals this with an ACK and the relay application sends the size of the soon-to-be-transferred data in `unsigned short int` (two bytes), which is again accepted by an ACK from the kit. Please note, that as both the MSP430 microcontroller and the x86 PC are little-endian, this multibyte numerical value doesn't need to be converted.

Because in the FITkit the communication is realized with the libfitkit *uart* module with relatively small receive (and send) buffers, to prevent data loss from buffer overflow, the data is split into chunks, 100 bytes each (save for the last one, which can be of course smaller). Every chunk must be ACK-ed by the kit before the transfer continues.

Note that the read request opcode is depending upon the usual behavior of most web browsers of sending the whole request in one piece, so that only one `recv()` system call is able to read the whole request message. This means that the commonly used HTTP server debugging tool, telnet, cannot be used with this implementation (except maybe the line-by-line version).

Waiting too long for a data transfer to occur may trigger a timeout (by the FITkit).

The transfer of 290 bytes of data, from issuing the READ_X command until the last ACK, in 12 steps, can be seen on Fig. 7:

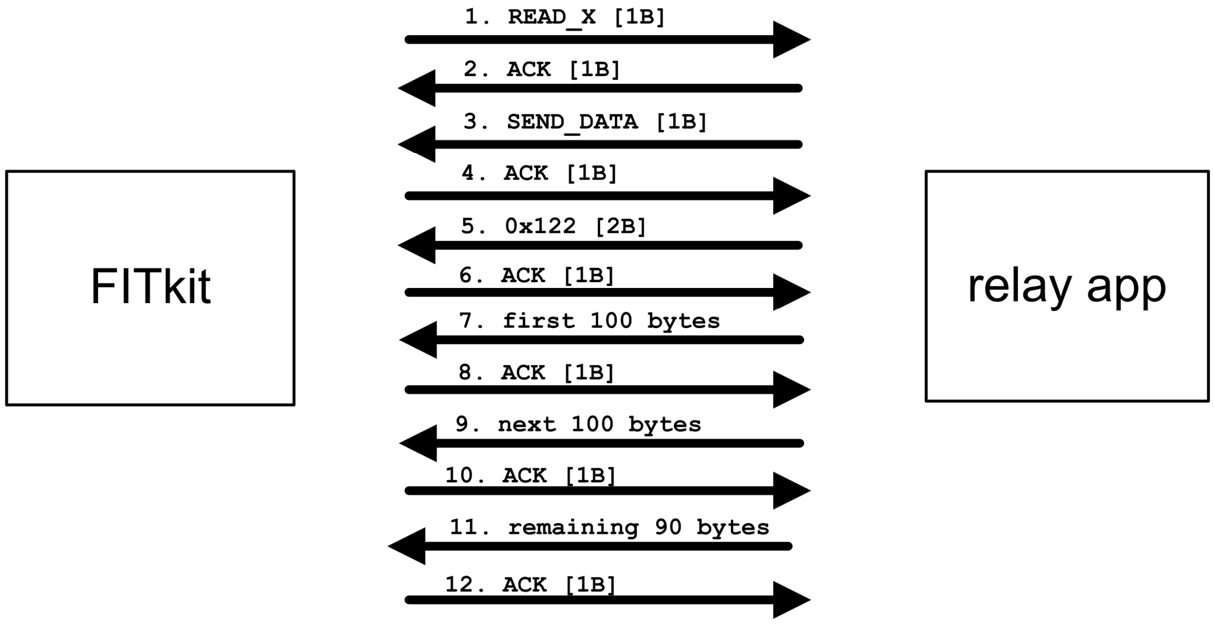


Fig. 7: Sample for transferring 290 bytes of data in 12 steps, after READ_X reception

4.2.5 Send data

This command instructs the relay application to accept the following data from the HTTP server and forward it to the client. It is handled by the `recv_datafromfitkit()` function.

First, the command is acknowledged, and then two bytes are read from FITkit, representing the size of the payload data in `unsigned short int`. Because the buffers on the computer are large enough to prevent data loss (unlike in the opposite direction), the payload is transferred in a continuous burst of bytes. In the next step the function tries to send the payload to the connected peer and informs the kit about the result with an ACK or NACK.

4.3 I/O subsystem

The I/O subsystem was created with the idea during the development to allow the easy change of the underlying input/output subsystem to TCP/IP if it becomes available for FITkit. According to this general rule, the HTTP server itself doesn't make any low-level input/output; every communication is through the higher level API calls.

I created a simple interface to use as a universal network communication API for the HTTP server. I tried to stay minimal and hide as many aspects of classical socket handling as possible (this of course might lead to performance or efficiency issues). The interface is defined in the `io_http.h` header file and the serial protocol implementation is in the `io_serial.c` file.

After taking the aspects of classical socket programming and information about the HTTP protocol into consideration I chose to have the following functions:

4.3.1 Function init_network

It’s a general network initialization function. In an Ethernet implementation this could be used as a placeholder for initializing the transceiver, creating the data layer connection, or registering for network communication in a multitask environment.

`io_serial.c:` In the serial implementation this is used to perform the initial handshaking, using the ACK opcode. To allow for some flexibility, it allows a few timeouts before failing. The transfer and receiving is done by calling the `term_send_char()`, `term_rcv_char()` and `term_rcv_char_t()`.

4.3.2 Function close_connection

As the HTTP protocol was built for a connection oriented protocol, and particularly older versions are using the closing of the connection for signaling the end of transfer (while newer ones are against this concept), this function was a must; it allows for implementing multiple version HTTP servers.

`io_serial.c:` This is a fairly easy task using the communication protocol between the computer and the FITkit. It consists of sending the CLOSE_CNN opcode and waiting for the ACK sent by the relay application.

4.3.3 Function accept_connection

For the same reasons as the close_connection function, this must be implemented as well.

`io_serial.c:` Again, using the relay application and the translator protocol to communicate with it, the hard part of this function is handled by the PC. But just to be sure we are on the safe side, this particular implementation first closes any active connection with sending a CLOSE_CNN command, waiting to be acknowledged and only then issuing the ACCEPT_CNN opcode. From this point, the successful accepting of a TCP connection results in the communication example seen on Fig. 6:

Because a connection might not come for some time, this function doesn’t time out waiting for one (although slow closing might force that).

4.3.4 Function recv

This function is intended for, compared to the previous ones, low-level manipulating with data: like the name suggests, it receives data from the network. In case of the HTTP protocol a higher level is also possible: receive whole request.

`io_serial.c:` In fact, the serial implementation assumes that a client will send the whole HTTP request in one piece and presents it to the server according to this: a complete request.

The communication between the I/O interface and the relaying application was discussed in detail before (see 4.2.4) with the associated example on Fig. 7: Unlike the accept_connection function, this function doesn’t wait indefinitely for data.

4.3.5 Function send

Finally, the send function: this allows the HTTP server to send back the response to the client.

`io_serial.c:` Unlike the send function this is implemented with no drawbacks and because the buffers are not holding it back (no need for data fragmentation), it is slightly faster then the receive function.

4.4 HTTP Server

After describing the inner workings of the relay application, the properties of the communication protocol used between FITkit and the PC, and what I/O interface was created for the HTTP server, it’s time to take a look how the server itself works.

It was developed in C language in conjunction with the network interface, using building blocks from the libfitkit library, mainly using its simple I/O interface (uart module) for sending and receiving application data over the USB to the PC.

The server (and the I/O subsystem) is using the following components of the FITkit:

- the MCU: handles all application and communication logic
- Channel B of the USB - serial converter (see 2.2.1.3) : for receiving requests and sending responses to the relay application running on a PC
- LCD display: status and error messages
- Indirectly also uses the FPGA chip and flash memory components (LCD controller)

The structure of the program is:

4.4.1 Initialization

This part is fairly short: because it is using the libfitkit, most of the work is taken care of by calling the initialize_hardware() function provided by it.

As the next step it starts the network initialization function, which, briefly, verifies that the communication channel is up and the relay application is working (for more information, see 4.3.1).

4.4.2 Main loop

The basic function of the main loop is to have the server application receive HTTP requests and provide the responses, indefinitely (or if an error occurs).

More on the following functions can be seen in the previous chapters, at their respective module description.

1. accept incoming connections by calling `accept_connection()` from the network API
2. preparing the global buffer for incoming requests
3. calling the `recv()` function to receive the client request
4. parsing the request to determine how to proceed and what response to send or generate
5. sending back the request through the PC with the `send()` function
6. closing the TCP channel with `close_connection()`

4.4.3 Parsing and generating responses

This implementation supports two out of three main request methods: the GET and HEAD. (Unfortunately the conditional GET method is not supported, because the FITkit lacks a clock and is unable to process the If-Modified-Since field.) Both methods have the same resource list for which they can be applied to, namely:

- The main page, with seven alias names for the resource URI ("/", /main.htm, /main.html, /index.htm, /index.html, /default.htm, /default.html
- The /favicon.ico icon for the address bar
- The /fitkitlogo.png: title picture in the main page

For these resources a valid 200 OK request is generated by the `create_response()` and `page_index()` functions.

First, the `create_response()` function generates the first part of the message, taking as arguments the status number (200), the MIME content type, (e.g., text/html) and the length of the entity, adding the constant HTTP version (1.0), and the server field.

For example, the response line and fields for the main page is the following:

```
HTTP/1.0 200 OK
Server: fitkit/0.01
Content-Type: text/html
Content-Length: 1009
```

In the next step, the `page_index()` function copies the rest of the message (the entity, in case of a GET request, or nothing in case of HEAD). This function also calls the `thermometer_gettemp()` function to determine the microcontrollers actual temperature and after converting it into a string, copies it into the HTML source code before sending. (the thermometer_gettemp() function is part of the libfitkit library).

In case of the much bigger *fitkitlogo.png* resource, which doesn't fit into the limited RAM, the response is sent in two parts: the output of the `create_response()` function and the resource.

Invalid URI

For invalid URI requests (not part of the previous list of resources) a 404 Not Found error response is sent. Again, the `create_response()` function generates the header of the response, given the status code.

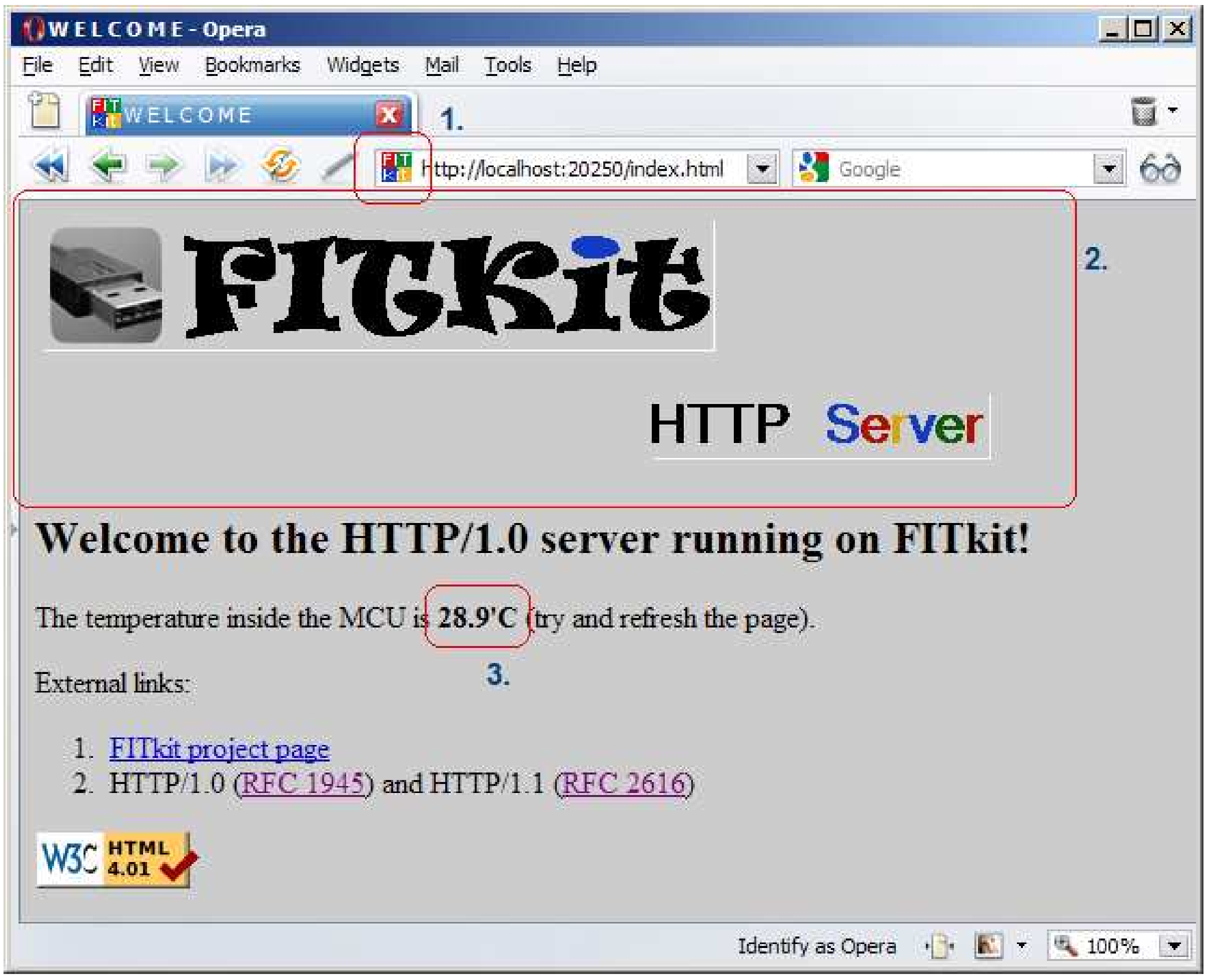
POST

Because this method was not implemented in this HTTP server, the server responds with the 501 Not Implemented status code.

4.4.4 Example

The main page hosted on the FITkit (Fig. 8:) with the following elements:

1. the favicon.ico
2. the fitkitlogo.png
3. the dynamic value of the internal MCU temperature


WELCOME - Opera
File Edit View Bookmarks Widgets Mail Tools Help
WELCOME
1.
http://localhost:20250/index.html
Google
2.
FITKit
HTTP Server
3.
Identify as Opera
100%

Welcome to the HTTP/1.0 server running on FITkit!

The temperature inside the MCU is 28.9'C (try and refresh the page).

External links:

1. FITkit project page
2. HTTP/1.0 (RFC 1945) and HTTP/1.1 (RFC 2616)

Fig. 8: The web page hosted on the FITkit

5 Conclusion

The aim of this work was to explore what obstacles arise when trying to implement the HTTP protocol (in one form or another) using the controllers and communication channels available on the standard FITkit device, and if possible, solve these issues.

First, the decision was made to implement the server side of the protocol, specifically creating a simple web hosting server, based on the fact that this would mean a huge client basis in form of countless web browsers, right after implementing the server application, besides the fact that hosting requires less RAM.

In the process of discovering the possibilities of connecting the designed HTTP server to its future clients, we came across new questions regarding available communication methods. Most of the web browsers are communicating via the TCP/IP protocol, but our teaching platform doesn't have the TCP/IP stack implemented, nor has a data link layer upon which could have been easily implemented.

The issue of connecting the HTTP server and clients was solved by using the USB interface connecting the teaching platform to the computer and developing a communication protocol able to transfer not only the HTTP requests and responses, but also controlling a relaying application running on the computer. The relay applications job is to "translate" between the client's native TCP/IP protocol and the protocol we just created.

During the final chapter we discussed the inner structure and workings of the mentioned relay application, talked about implementation details of the devised USB-TCP/IP socket converting protocol, the I/O interface used by the HTTP server and the server itself: how does it handle incoming requests, the request line parsing, response compiling details, the sample web page it is hosting and its elements.

Although, as we saw, the HTTP protocol doesn't appear to be a very complicated protocol, it does have properties which are difficult to implement in an embedded system: it is completely text-based and the message lengths are variable. Parsing strings with variable length is usually a memory-intensive task and this is the very bottleneck in a standard FITkit device and many implementation restrictions were made to create the server.

In further work the SDRAM memory controller should be definitely included which would provide substantially more storage for parsing HTTP requests, generating dynamic responses, etc. Another viable extension would be incorporating an Ethernet module with TCP/IP stack and implement the newer HTTP/1.1 protocol version.

References

[1] Texas Instruments Incorporated. *Datasheet for MSP430F241x, MSP430F261x mixed signal microcontroller* [online]. 2007, JANUARY 2009 [cit. 2009-05-05]. Dostupný z WWW: <http://focus.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=slas541e&fileType=pdf>

[2] VAŠÍČEK, Zdeněk, , et al. *FITkit (platform homepage and documentation)* [online]. 2006-2009 [cit. 2009-05-15]. Dostupný z WWW: <http://merlin.fit.vutbr.cz/FITkit/ >

[3] Xilinx. *Spartan-3 Generation FPGA User Guide : Extended Spartan-3A, Spartan-3E, and Spartan-3 FPGA Families* [online]. 2006 , UG331 (v1.5) January 21, 2009 [cit. 2009-05-10]. Dostupný z WWW: <http://www.xilinx.com/support/documentation/user_guides/ug331.pdf>.

[4] BERNERS-LEE, T., FIELDING, R., FRYSTYK, H.. *Hypertext Transfer Protocol -- HTTP/1.0* [online]. 1996 , May 1996 [cit. 2009-05-05]. Dostupný z WWW: <http://www.rfc-editor.org/rfc/rfc1945.txt>.

[5] FIELDING, R., et al. Hypertext Transfer Protocol -- HTTP/1.1 [online]. 1999- , June 1999 [cit. 2009-05-05]. Dostupný z WWW: <http://www.rfc-editor.org/rfc/rfc2616.txt>.

[6] MOGUL, J. C. , et al. Use *and Interpretation of HTTP Version Numbers* [online]. 1997 [cit. 2009-05-05]. Dostupný z WWW: <http://www.rfc-editor.org/rfc/rfc2145.txt>

[7] BERNERS-LEE, T.. *The Original HTTP as defined in 1991* [online]. 1991 [cit. 2009-05-05]. Dostupný z WWW: <http://www.w3.org/Protocols/HTTP/AsImplemented.html>

[8] STEIN, Greg. *DAV Frequently Asked Questions* [online]. 2000 , Wed Oct 11 03:46:50 PDT 2000 [cit. 2009-05-15]. Dostupný z WWW: <http://www.webdav.org/other/faq.html>

[9] FRANKS, J., et al. *HTTP Authentication: Basic and Digest Access Authentication* [online]. 1999 [cit. 2009-05-05]. Dostupný z WWW: http://www.rfc-editor.org/rfc/rfc2617.txt

[10] Microsoft Corporation. *Getting Started with Winsock* [online]. 2009 , Build date: 5/14/2009 [cit. 2009-05-10]. Dostupný z WWW: <http://msdn.microsoft.com/en-us/library/ms738545(VS.85).aspx>

Attachment list

Attachment 1: Augmented BNF, [4]
Attachment 2. Source codes on CD.

ATTACHMENT 1

```
The augmented BNF includes the following constructs:

 name = definition

     The name of a rule is simply the name itself (without any
     enclosing "<" and ">") and is separated from its definition by
     the equal character "=". Whitespace is only significant in that
     indentation of continuation lines is used to indicate a rule
     definition that spans more than one line. Certain basic rules
     are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc.
     Angle brackets are used within definitions whenever their
     presence will facilitate discerning the use of rule names.

 "literal"

     Quotation marks surround literal text. Unless stated otherwise,
     the text is case-insensitive.

 rule1 | rule2

     Elements separated by a bar ("I") are alternatives,
     e.g., "yes | no" will accept yes or no.

 (rule1 rule2)

     Elements enclosed in parentheses are treated as a single
     element. Thus, "(elem (foo | bar) elem)" allows the token
     sequences "elem foo elem" and "elem bar elem".

  *rule

    The character "*" preceding an element indicates repetition. The
    full form is "<n>*<m>element" indicating at least <n> and at
    most <m> occurrences of element. Default values are 0 and
    infinity so that "*(element)" allows any number, including zero;
    "1*element" requires at least one; and "1*2element" allows one
    or two.

  [rule]

     Square brackets enclose optional elements; "[foo bar]" is
     equivalent to "*1(foo bar)".

   N rule

     Specific repetition: "<n>(element)" is equivalent to
     "<n>*<n>(element)"; that is, exactly <n> occurrences of
     (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a
     string of three alphabetic characters.

 #rule
```

A construct "#" is defined, similar to "*", for defining lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by one or more commas (",") and optional linear whitespace (LWS). This makes the usual form of lists very easy; a rule such as "(*LWS element *(*LWS "," *LWS element))" can be shown as "1#element". Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element), , (element)" is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element must be present. Default values are 0 and infinity so that "#(element)" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

; comment

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

implied *LWS

The grammar described by this specification is word-based. Except where noted otherwise, linear whitespace (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent tokens and delimiters (tspecials), without changing the interpretation of a field. At least one delimiter (tspecials) must exist between any two tokens, since they would otherwise be interpreted as a single token. However, applications should attempt to follow "common form" when generating HTTP constructs, since there exist some implementations that fail to accept anything beyond the common forms.

Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII coded character set is defined by [17].

```
OCTET          = <any 8-bit sequence of data>
CHAR           = <any US-ASCII character (octets 0 - 127)>
UPALPHA        = <any US-ASCII uppercase letter "A".."Z">
LOALPHA        = <any US-ASCII lowercase letter "a".."z">

ALPHA          = UPALPHA | LOALPHA
DIGIT          = <any US-ASCII digit "0".."9">
CTL            = <any US-ASCII control character
                 (octets 0 - 31) and DEL (127)>
CR             = <US-ASCII CR, carriage return (13)>
```

```
    LF             = <US-ASCII LF, linefeed (10)>
    SP             = <US-ASCII SP, space (32)>
    HT             = <US-ASCII HT, horizontal-tab (9)>
    <">            = <US-ASCII double-quote mark (34)>
```

HTTP/1.0 defines the octet sequence CR LF as the end-of-line marker for all protocol elements except the Entity-Body (see Appendix B for tolerant applications). The end-of-line marker within an Entity-Body is defined by its associated media type, as described in Section 3.6.

```
    CRLF           = CR LF
```

HTTP/1.0 headers may be folded onto multiple lines if each continuation line begins with a space or horizontal tab. All linear whitespace, including folding, has the same semantics as SP.

```
    LWS            = [CRLF] 1*( SP | HT )
```

However, folding of header lines is not expected by some applications, and should not be generated by HTTP/1.0 applications.

The TEXT rule is only used for descriptive field contents and values that are not intended to be interpreted by the message parser. Words of *TEXT may contain octets from character sets other than US-ASCII.

```
    TEXT           = <any OCTET except CTLs,
                      but including LWS>
```

Recipients of header field TEXT containing octets outside the US-ASCII character set may assume that they represent ISO-8859-1 characters.

Hexadecimal numeric characters are used in several protocol elements.

```
   HEX            = "A" | "B" | "C" | "D" | "E" | "F"
                    | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT
```

Many HTTP/1.0 header field values consist of words separated by LWS or special characters. These special characters must be in a quoted string to be used within a parameter value.

```
       word           = token | quoted-string


       token          = 1*<any CHAR except CTLs or tspecials>

       tspecials      = "(" | ")" | "<" | ">" | "@"
                      | "," | ";" | ":" | "\" | <">
                      | "/" | "[" | "]" | "?" | "="
                      | "{" | "}" | SP | HT
```

Comments may be included in some HTTP header fields by surrounding the comment text with parentheses. Comments are only allowed in fields containing "comment" as part of their field value definition. In all other fields, parentheses are considered part of the field value.

```
    comment        = "(" *( ctext | comment ) ")"
    ctext          = <any TEXT excluding "(" and ")">
```

A string of text is parsed as a single word if it is quoted using double-quote marks.

```
    quoted-string  = ( <"> *(qdtext) <"> )

    qdtext         = <any CHAR except <"> and CTLs,
                     but including LWS>
```

Single-character quoting using the backslash ("\") character is not permitted in HTTP/1.0