TCPIP Stack - Manual

Document ID:	TEA0031-002
Status:	Released
Version:	1.1
Date:	2004-04-08

A L T I U M B V

Contents

Revision History

Version	Date	By	Reason	Approved
1.0	November 19, 2002	TEA	Initial version	December 3, 2002
1.1	April 8, 2004	TEA	Software revision 1.1	April 19, 2004

1 Introduction

1.1 Basics

This stack is a minimalistic implementation of the IP/UDP/TCP protocols. Small code size and minimal RAM usage were the major requirements; this is not a full-fledged implementation of the defining RFC documents.

Included with the stack is a SLIP/PPP implementation with support for Hayes compatible modems, capable of dialing in to commercial internet providers, or receiving calls from out-of-the-box PPP clients. To facilitate the dial-in process a basic DNS server is running, so users can browse to a hostname instead of an IP-number.

An alternative transport layer is Ethernet, with basic DHCP and ARP support.

An HTTP server is included. Although basic it should be sufficient for a lot of implementations. Its most distinguishing ability is to create a simple CGI-interface to chosen C-variables, based on a single definition line for each variable. It even optionally generates a human usable HTML-form on top of the same CGI interface. By default is serves pages from hardcode byte arrays.

An FTP server allows access to a FAT-based filesystem; the same filesystem is optionally available to the HTTP server.

Finally a TELNET server, a DNS client and a SMTP client are present. The TELNET server is only a skeleton demo but can be used as a starting point for further development; the combination DNS/SMPT client is directly usable to send mail from a device to the outside world.

Last but not least: all features can be included on an "as needed" basis, where functionality not included is not compiled.

1.1 Simple setup for HTTP-based configuration or reporting with existing or new projects

One of the more outstanding features of the stack is the HTTP server included, with its built-in CGI capabilities and full GUI-tool support.

It basic use is to create a browser-based interface to a new or existing device. With the GUI-tool supplied simple variables can be exported for reading from or writing to an embedded device.

User supplied code could expand on the basic variable-types included with the stack, for instance to have it send a camera image over the CGI-interface supplied.

1.2 Graphical setup tool & template source

A GUI-based setup tool is included to make configuring the stack a breeze.

Template source for projects is present to get you started. A separate chapter in this manual walks you through using these templates building a new project, a new server or a new client.

Also included is template source for TCP or UDP servers if you need to implement your own dedicated service.

1.3 Single buffer, zero copy

The stack uses a single buffer for input, output and all processing. Contrary to usual implementations all operations are carried out in place, instead of copying data when transferring a packet to or from another level in the stack. When replying, often headers are reused by swapping incoming/outgoing ports and addresses while leaving other bytes untouched.

As a side-effect this means that every incoming packet has to be fully processed and responded to before the stack can start processing the next incoming packet. Which in an embedded system even might be a Good Thing; this guarantees the only buffer overflow can take place at the lowest level at the transport layer where sometimes a handshaking mechanism is available to slow down the other side.

(An exception to the in place accessing above is the accessing of headers on some platforms. As some processors can only (or only efficiently) access word-aligned data, on these platforms the headers are mirrored in separate buffer-structures.)

1.4 Non-standard interface

No BSD-like socket interface is implemented, only a simple call-back based mechanism. This means the stack can run without an RTOS and memory requirements are stable and predictable.

The servers themselves are responsible for rebuilding data if a resend is requested, this avoids having to buffer old data. If a minimal solution is wanted retries can just be rejected, which in practice is very usable on a single serial dial-in connection.

The main entry of the stack should be called as often as possible. To control system load however a basic timesliced RTOS is advisable, as no statements are made regarding time spend in a call to the main entry of the stack. Be advised however that any servers or clients connected to (and thus called by) the stack will be running in the same thread.

A timer entry should be called regularly, about once a second is OK (based on the default timeouts). Irregularities are no problem; the only function of the timer entry is to count down several timeouts which are defined in ticks (where one 'tick' is one call to the timer entry). None of the timings is critical and all are implementable within a very wide range to accommodate different environments.

1.5 Minimalistic interpretation of the RFCs

Every protocol is implemented in a minimalistic manner to minimize ROM and RAM requirements. This means no RFC is implemented front to back, but instead only the basics used by tested real-world implementations are supported.

An example is PPP negotiations. Although per spec every separate layer could be taken down and re-negotiated up again, no real-world PPP client starts this process by itself. The PPP handshaking build into the stack is geared to connecting each layer in turn, and doesn't recognize taking down a layer. If a fictitious client would ask to take a layer down, it would get a reject of the command, after which it would probably abort the whole link and report a communication error.

This is the type of behaviour what has been tried to achieve with the stack: the protocols are implemented incompletely, but either this is ignorable, or it can be detected by the other side of the connection.

The following protocols are thus implemented:

- HAYES modem support
- Support for Microsoft's NTRAS direct serial connection
- SLIP

- PPP (LCP, PAP, IPCP, HDLC)
- IP4
- ICMP
- TCP
- UDP
- DNS (UDP-based) client & server
- HTTP server
- FTP server
- SMTP client
- TELNET server

1.6 Overview of some major implementation shortcuts:

1.6.1 No IP-address checking for serial connections

The stack itself ignores all IP-addresses. Every packet is responded to with its addresses swapped. For a normal client this is no problem as this only means they see the device as an unlimited collection of identical servers. Their routing table is responsible which requests are sent to their serial link and thus will show up our device.

Individual servers are free to implement a stricter filtering if needed.

1.6.2 Very minimal ICMP implementation

Only ECHO is accepted (ECHO REPLY is returned) and only PORT UNREACHABLE is ever send. All other incoming packets are dropped.

1.6.3 TCPIP retries will only resend the last packet

In a series of IP packets each packet is only send when the previous is acknowledged. In theory when A/B/C is send and C is acknowledged, an old repeated acknowledge for A could arrive again. Normal behaviour should be to resend B. Our stack is only capable of retransmitting C, if asked otherwise the session will be terminated by sending a RESET instead.

1.6.4 Very minimal DNS server

The server included can answer one question: resolve <our-hostname> into <our-IP-address>. All other questions are ignored, and the answer is returned with a few minutes lifespan.

1.7 Filesystem

To store the files needed, two filesystems are supported: a read only filearray (usually in ROM), and a writable one on any device (for now only a lowlevel RAM driver is included). For both filesystems a PC based tool is supplied to generate a "disk image" from a file-tree located on the PC.

1.7.1 Hardcoded HTTP url/file array

This is simply a list of file-URLs with their contents in bytearrays. It is only supported by the HTTP server.

1.7.2 FAT-based

Included with the stack is a FAT-based filesystem. A lowlevel driver for it to run from an image in memory is included as well.

The FTP server runs exclusively on top of this filesystem, for the HTTP server its use is optional.

2 The stack from a BSD-socket perspective

2.1 Introduction

This chapter tries to give an introduction to the way the stack works, while addressing prominently the places where the stack and its server/clients behaviour deviates from classic BSD-socket implementations.

Very simply put a classic BSD-socket type of stack has processing like:

```
Loop:
      For any incoming frame
            Put the frame in the correct socketbuffer
      For any socket buffer
            Send any pending outgoing frames
      For any socket buffer with a timeout
            Resend the last frame
```

Apart from this each server/client in its own thread will run:

```
Loop:
      optionally: read from the socket
      do any processing
      optionally: write to the socket
```

The Altium stack however uses the following flow:

```
If any new frame is received:
      Call the correct server/client-function with the frame
      Server/client processes the frame, creating a reply-frame
      Send back the reply-frame
Else:
      For any session which has generated new data:
            Send the data
      For any session which has timed-out:
            Call the correct server/client-function asking for a resend
            Server/client processes resendrequest, recreates the frame
            Send the frame
```

Two flow-related observations here:

- The server/clients are not running in separate threads
- The stack itself does not loop and must be called repeatedly in a loop from the user-program.

Both issues are related to having the stack run without the need of an RTOS.

2.2 Servers are installed compiletime

Although it is possible to add them runtime, normal use of the Altium stack has all combinations of available TCP-serverports and pointers to their serverfunctions defined together in a compiletime filled list. There is no other user written code needed to have a server 'listening' on its port.

2.3 Stack calls servers/clients

With BSD-sockets each server is polling or waiting on the stack until there is a new connection. At that moment a new socket is opened and a thread is split off to handle the new session.

With the Altium-stack the stack handles incoming connections, finds out which server or client is listening on that port and calls the server/client function.

2.4 Server/clients are aware of TCP-sessions

A server gets called whenever a new TCP frame is coming its way. The server/client function is able to read and/or set TCP flags in the header to influence the flow of the session.

Although this may sound intimidating, first of all when using the included servers they can be considered as a black box and used as-is. Secondly when building a new TCP server there are is chapter included in this document how to build a new server starting from the sample-source (templates) included with the stack.

2.5 Server/clients are aware of TCP-frames

This means a server is always aware of how much data came in with a frame. It *must* process all incoming data and it *must* limit the size of its answer to the maximum allowed data in a TCP frame.

A server receives all frames as soon as they come in, and the stack will only continue processing as soon as the reply has been given.

With a BSD-socket interface a server/client can always send or wait for as much data as it wants, the stack will send out multiple frames if needed, or wait for reception of multiple frames until the required amount has been received.

2.6 Server/client must re-create its last answer upon request

The stack itself uses no buffers at all apart from one single working buffer. This means it cannot respond on its own to any request for retransmission of the last TCP-frame send.

In these cases the stack will call the server with a request for re-writing the frame of data send by the server.

2.7 Setting up a client

A client is communicated runtime to the stack to handle a certain portnumber while its TCP-session lasts. It is initiated by user-written code calling tcp_create().

The client function can either be one included with the stack (SMTP) or one written by the user. When building a new TCP client there are is chapter included in this document how to build a new server starting from the sample-source (templates) included with the stack.

3 Using the stack

3.1 Introduction

The stack includes a set of configuration tools to facilitate adding a TCPIP stack to your projects.

A GUI tool gives easy access to all configuration parameters and writes your settings in a human-readable ini-file. A set of commandline tools create all include- and source-files needed for your project, based on the choices you stored in the ini-file. Also included are tools to create diskimages from a tree of files on your development machine.

The GUI tool calls the commandline tools when needed, but commandline access is available in case you want to include the stack in an automated build process.

3.2 QuickStart

The following shows how to get a simple project up and running.

This example lets you create a HTTP server device. When it is finished you can connect from you development machine (with PPP dialin over a direct cable connection) to the embedded device and use any browser on the development machine to view the pages stored on the device.

In the example {platform} is your hardware platform, {myproject} is the name of your example project and {tcpip} is the location where the TCPIP-stack is installed.

- Create a workdirectory {myproject} somewhere
- Create a directory {myproject}\httpfiles. Add a few small example files in this directory you want served from the device. Be sure to add at least an 'index.html'.
- Copy {tcpip}\templates\template_main.c to {myproject}\mymain.c.
- Copy {tcpip}\{platform}\template_tcpipsysset.h to {myproject}\tcpipsysset.h
- If you want you can check the template defines in the mymain.c and tcpipsysset.h to see if the settings are workable for you, they should normally suffice for this example.
- Start {tcpip}\tools\stackconf.exe. Select the 'Transport' tab, uncheck 'Modem support'
- Select the 'Servers' tab. Enter 'httpfiles' in the field 'Hardcoded files root (host)'.
- Select menu 'Action', choice 'Generate .c/.h/.bin'. The tool will ask you to save first, choose your {myproject} directory and save under any name (no extension). Important: do ***not*** save in any other place, because the generated files will end up in this directory.
- Start your EDE, create a new project in {myproject}, add 'mymain.c'.
- Open {myproject}\tcpipfiles_readme.txt. It tells you what files to include in your project, both from the TCPIP stack as well as generated by the configuration tool. Don't forget to set the include directories as described at the top of the file.
- Build, flash or download, reset.

You now have a working HTTP server device with PPP call-in facilities.

Use for instance the 'Dialup Networking' facilities from Microsoft Windows to test the device. Select 'PPP' for the 'Dial server type', and 'Dialup networking serial cable between 2 PCs' for 'Dial using'. The user and password (if you didn't change the template defaults in mymain.c) are 'user' and 'password'.

You can now use any webbrowser to go to '192.168.100.1' (again: if you didn't change the template defaults in mymain.c). The 'index.html' you put in {myproject}\httppages\index.html is then served by the device and shown in your browser.

4 The configuration tools

4.1 stackconf.exe - the GUI to all settings

Lets you edit the project-ini where all stack related settings are kept.

4.1.1 Field 'Custom layout'

This specific commandline option (or field 'Entry Format' if using the GUI) needs a bit of explaining. It is only needed for very few compilers in rare circumstances, normally it should be left unchanged.

The makediskimg tool has to build a memoryimage of the disk. The tool runs on the PC, but the diskimage will be used by the embedded code. The tool however has no knowledge of the way the embedded C-compiler used will allocate its FAT structure in memory.

With nearly all compilers knowing the endianness and alignmentsize gives enough information to predict how the FAT structure will end up in memory. The 'Entry format' setting however can override the defaults, in case a specific use of a specific compiler makes it allocate structure-elements in a non-standard way.

The setting first describes the layout the compiler will use to allocate a FAT entry in memory. The FAT-structure as used internally by the stacks filesystem is defined as:

```
{
    Uint32 filesize;               // 'F' in Entry Format
    Uint16 cluster;                // 'C' in Entry Format
    Uint8 status;                  // 'S' in Entry Format
    char name[FILENAMELENGTH];     // 'N' in Entry Format
}
(where FILENAMELENGTH is userspecified with commandline or GUI)
```

The first character in the format defines little-endian ('L') or big-endian ('B'), the other characters describe byte-by-byte what entry is located at that position. Unused bytes are indicated by a '0' (zero).

A few examples:

- A little-endian byte aligned compiler with filenamelength 9: "LFFFFCCSNNNNNNNNN"
- A big-endian 2-byte aligned compiler with filenamelength 7: "BFFFFCC0SNNNNNNNN0"

Note: these two examples in fact describe the default allocation for the given endianness and alignment-size. The format described for them is the default behaviour of the makediskimg tool for these endianness/alignment combinations. The 'Entry Format' setting didn't have to be used in these specific examples!

4.2 stackconfmake.exe - generate sourcecode based on your settings

Called by stackconf.exe to generate the header with settings for your project. It even generates a readme describing which files from the TCPIP stack are needed in your project based on your settings.

Calls makediskimg.exe and makefixedfs.exe with the correct parameters (only if you used the GUI to select a tree with files for either HTTP serving hardcoded data or initial content of a live filesytem).

For automated builds stackconfmake.exe can be called from the commandline with your ini-file as single parameter to generate all files needed for your project.

4.3 makediskimg.exe - generate sourcecode for initialized diskimages

Called by stackconfmake.exe. It generates an initial diskimage for the live filesystem included with the TCPIP stack. Optionally it takes a tree of files on your development station to populate the filesystem. If not, the image will just be preformatted blank.

4.4 makefixedfs - generate sourcecode for hardcoded HTTP-data

Called by stackconfmake.exe. It takes a tree of files on your development station and generates a source which can be used by the HTTP server to serve hardcoded files.

5 Demos & 'Getting Started Templates'

5.1 Demonstration projects included

Two complete projects are included as examples. The first uses about as much features as possible, the second is a very much stripped version only serving a few CGI variables with a HTTP server.

Both demos at least consist of:

- a temperature sensor (just a potentiometer on the SDK or a fake slow sawtooth if none available)
- a user-controllable threshold value
- a 'yellow' led (connected to a real led if one is available) switching on whenever the 'temperature' is under the 'threshold'
- a user-controllable 'red' led (connected to a real led if one is available)

5.1.1 Demo 'General'

If an ethernet driver is available for the platform this demo will use it (without DHCP client but using a fixed IP-address), otherwise it will use PPP over a serial connection with the DNS server (dial in with user 'user' and password 'password').

You can TELNET to the demo or use an FTP client (both with user/user, test/test or root/root as username and password).

An HTTP client can be used, pages are served for showing of SSI, setting CGI variables and serving both hardcoded pages and using the FAT filesystem. If you use FTP with user 'root' to upload a page to the directory /htdocs, it will be available from the webserver.

The stackconf-tool can be used to play around with any setting, and as there are #ifdef-tests all over the source most changes will work without sourcecode modifications. Only when adding or removing servers or changing transports the relevant sources should be added to or removed from the project.

5.1.2 Demo 'HTTP-CGI'

This demo is a very stripped down version of the general demo, you can only call in using PPP (modem or direct connection with user 'user' and password 'password').

Only an HTTP client can be used, and only the sample form generated by the stackconf-tools is being served with the demo variables.

5.2 'Getting Started Templates' included

To get you started besides the demos a number of templates have been created. Some of the are usable straight away (especially the template_main.c), others are real skeletons only giving you a point to start coding yourself (TCP en UDP servers).

For an elaborate example of how to use these templates see chapter 16: 'Creating your own TCP client or server'.

5.2.1 template_main.c

This template is usable out of the box to jump start a new project. Compare it with the demos to see how to integrate an existing project.

5.2.2 template_ppp_smtp_dns.c

This template shows how to integrate a DNS client, an SMTP client and PPP client to create a device which can automatically dial a provider, resolve a mailservername into its IP-address, send a mail, and break the connection.

The triggers are based on fictitious keyboard input, but they show the way how to initiate certain actions.

(The template is not meant to be copied entirely, but parts of it can be re-used.)

5.2.3 templace_tcp.c/.h

If you need a special TCP server or client for your project this skeleton is the starting point. At the end of this manual you can find an appendix describing the userclient/userserver demo's included with the stack.

5.2.4 template_udp_server.c

If you need a special UDP server for your project this skeleton is the starting point.

5.2.5 template_telnet_commandprocessor.c

The TELNET server included with the stack has no usable commands as-is, they will be fully defined by the needs of your project. This skeleton can be used to start developing a commandprocessor capable of triggering those actions you want protected by a password, possibly user-dependant.

Note that the TELNET server as supplied is not meant to return a lot of information, as its replies are limited to the datasize of the TCP frame. For bigger amounts of information the HTTP server or possibly the FTP server should be used.

6 API - General

6.1.1 Source

common\tcpip.c common\tcpip.h	All IP/ICMP/UDP/TCP code
common\sys.h	General hardware header, mostly prototypes and defaults for platform dependant code
<platform>\sys_<platform>*.c <platform>\sys_<platform>.h	Hardware related code, platform dependant
sys_rom.c	Used if a platform needs different code to access constant data
<project>\tcpipset.h	Project dependant settings

6.2 Compile time settings in tcpipset.h

6.2.1 #define OPTION_ICMP_ECHO

If defined a valid response to ICMP ECHO packages will be given.

6.2.2 #define OPTION_UDP

If defined support for the UDP transport protocol will be compiled in. Servers & clients have to be supplied by the user except for both a very basic DNS server & client which are supplied.

6.2.3 #define TCP_PORT_CLIENT_START <portnr>

(default 1000). See next paragraph.

6.2.4 #define TCP_PORT_CLIENT_END <portnr>

(default 2000)

TCP_PORT_CLIENT_* range is used for sessions initiated by the device by calling tcp_create() with -1 for the source-port.

6.2.5 #define TCP_RETRY_TICKS <ticks>

(time in ticks, default 5)

Time before retrying TCP transmits.

6.2.6 #define TCP_RETRY_MAX <ticks>

(default 3)

Number of times to retry TCP transmits before resetting the session.

6.2.7 #define TCP_WATCHDOG_TICKS <ticks>

(default 300)

Time we accept without activity on a session before resetting it.

6.2.8 #define TCP_LASTACK_TICKS <ticks>

(time in ticks, default hardcoded to 5)

Time we ignore trailing incoming packets on a closed session before rejecting them by returning a RESET.

6.2.9 #define TCP_SESSIONS <sessions>

(default hardcoded 16)

One session is used for each simultaneous TCP connection; the RAM cost is one TCPSESSION structure per session.

6.3 Compile time settings in tcpipsysset.h

6.3.1 #define USER_CLIENTSERVER[1-4]_H <headerfile >

6.3.2 #define USER_CLIENTSERVER[1-4]_CARGOTYPE <cargotype>

6.3.3 #define USER_CLIENTSERVER[1-4]_CARGO <cargoname>

These defines can be used to communicate four user-defined TCP clients or servers to the stack. If more are needed the common/tcpip.h must be changed by hand. For an example of the usage see chapter 'Creating your own TCP client or server'.

6.4 Runtime settings in global tcpip_settings structure

6.4.1 Uint8 tcpip_settings.self_ip[4]

Our own IP address (can be set to 0.0.0.0 if transport layer supports assigning, think PPP or DHCP).

6.4.2 TCP_SERVER tcpip_settings.tcpservers[]

List of TCP servers and the ports they listen on. The list must be terminated with a NULL-server.

Example:

```
// TCP_SERVER is defined as:
// struct
// {
//    Uint16 port,
//    TCP_SERVERFUNCTION serverfunction;
// }
//
static TCP_SERVER tcpservers[] =
{
```

```
      {80, http_server},
      {8080, http_server},
      {21, telnet_server},
      {1050, my_tcp_server},
      {0, NULL}
};
tcpip_settings.tcpservers = tcpservers;
```

The double occurrence of http_server shows how a tcp_server can listen on multiple ports.

6.4.3 UDP_CLIENTSERVER tcpip_settings.udpclientservers[]

List of UDP clients and UDP servers and the ports they listen on. The list must be terminated with a NULL-server.

Example:

```
// UDP_SERVER is defined as:
// struct
// {
//    Uint16 port,
//    UDP_CLIENTSERVERFUNCTION clientserverfunction;
// }
//
static UDP_CLIENTSERVER udpclientservers[] =
{
      {0, dns_client},
      {0, dns_client},
      {68, dhcp_client},
      {53, dns_server},
      {0, NULL}
};
tcpip_settings.udpclientservers = udpclientservers;
```

The double occurrence of dns_client shows how a client could listen on two ports simultaneously, for instance the dns_client could listen for the answers to two outstanding DNS requests. The user-written code is responsible for replacing the 0-portnumbers with the correct ports when initiating the request, and setting them back to 0 afterwards.

6.5 Usage

6.5.1 void tcpipstack_init(void)

Use once on startup to initialize all modules used. The tcpip_settings structure should be filled beforehand.

6.5.2 void tcpip_timertick(void)

Should be called approximately every second. This calls all other modules *_tick() functions, with the exception of dhcp_seconds_tick().

This function makes the stack verify all its internal resend-counters and session-watchdogs.

6.5.3 void tcpip_main(void)

Should be called as often as possible. If it returns 0 then the stack had nothing to do and the rest of the time slice could be yielded if running under an RTOS.

Incoming UDP, ICMP or TCP frames are automatically processed by the stack for those services defined, if needed the appropriate UDP, TCP server or TCP-client functions are called.

A TCP server or client will be called when there is any incoming communication on its port, or if it has to resend due to a timeout, or if any external source has set a tcp_session->resend value to -1.

A UDP server will be called when there is any incoming communication on its port.

6.5.4 TCP_SESSION *tcp_create(Uint16 source_port, Uint32 destination_ip, Uint16 destination_port, TCP_CLIENTFUNCTION clientfunction)

Initiate a session from the device to an outside TCP-server.

parameter	usage
source_port	TCP port embedded side. If -1 it will be automatically generated from the range TCP_PORT_CLIENT_START to TCP_PORT_CLIENT_END.
destination_ip	IP address remote side
destination_port	TCP port remote side
clientfunction	callback TCP client function
(returns)	pointer to session information structure, NULL if creation failed

Keep in mind this only sets up internal session-information for the stack. Only on consecutive calls to tcpip_timertick() and tcpip_main() the stack will actually start building the session.

6.5.5 void tcp_resetall(void)

Reset all open TCPIP sessions.

7 API - Serial transport

7.1 Source

common\serial.c common\serial.h	All serial/SLIP/PPP code
drivers\serial_driver.h	Lowlevel driver header, mostly prototypes and defaults for platform dependant code
<platform>\serial_<platform>.c <platform>\serial_<platform>.h	Lowlevel driver code (platform dependant)

7.2 Compile time settings in tcpipset.h

7.2.1 #define OPTION_SERIAL

If defined this turns on the SERIAL module to transport TCPIP over a serial (RS232) line.

7.2.2 #define SERIAL_MODEM

If defined the module supports modems, this can be used for SLIP or PPP connections.

7.2.3 #define SERIAL_SLIP

If defined the module supports the SLIP protocol to transport IP packages. The IP-address settings of the TCPIP modules are ignored, any incoming traffic is replied to with destination/source addresses reversed.

7.2.4 #define SERIAL_PPP

If defined the module supports the PPP protocol to transport IP packages. Depending on SERIAL_CALLIN / SERIAL_CALLOUT the connection can be made "as" a provider or "to" a provider. When dialing out both IP-addresses from the link are negotiated with the provider, when dialed in the IP-addresses from the TCPIP settings are used.

If OPTION_DNS_CLIENT is defined a DNS IP address is asked from the server when dialing out, if OPTION_DNS_SERVER is defined our DNS IP address is told to the client when dialed in.

7.2.5 #define SERIAL_CALLIN

If defined, the device can be used to dial in to. The module will answer a modem (if SERIAL_MODEM is defined) and if SERIAL_PPP is used all handshaking needed will be conducted.

7.2.6 #define SERIAL_CALLOUT

If defined, the device can be used to dial out with. The module will dial a number (if SERIAL_MODEM is defined) and if SERIAL_PPP is used all handshaking needed will be conducted.

7.2.7 #define SERIAL_BUFLEN <bytes>

(size in bytes, default hardcoded to 1500)

Length of PPP/SLIP buffer, excluding all HDLC line framing. The default of 1500 is sufficient for all normal PPP/SLIP connections. PPP has been successfully tested with a small buffer of 250 bytes.

7.2.8 #define SERIAL_PACKETTIME_TICKS <ticks>

(time in ticks, default hardcoded to 5)

Maximum time allowed for an incoming packet over the serial line, from the first header byte to the complete reception of the packet. If this time is exceeded while no new data is received, the incoming statemachine is reset and any pending received data is discarded.

7.2.9 #define SERIAL_MODEMCONNECTTIME_TICKS <ticks>

(time in ticks, default hardcoded to 45)

Maximum time allowed for modem dial-in or dial-out, from the initial ATDT or ATA command to the received CONNECT. If no CONNECT is received during that time the process is aborted and LINESTATE_IDLE reported through the callback interface.

7.2.10 #define SERIAL_PPPCONNECTTIME_TICKS <ticks>

(time in ticks, default hardcoded to 15)

Maximum time allowed to complete PPP handshaking, starting with the CONNECT received from the modem. During that time the LCP, PAP and IPCP handshaking should be finished (PAP single way, others two-way). If not, the line connection is broken with a LCP TERMINATE, and if a modem was connected the link is broken. LINESTATE_IDLE is reported through the callback interface.

7.2.11 #define SERIAL_MODEMBREAKTIME_TICKS <ticks>

(time in ticks, default hardcoded to 15)

Maximum time allowed for the modem to respond to a break. If no response is received a hang-up is send with the same timeout. If no response is received a break is send again with the same timeout. Repeat ad infinitum.

7.3 Runtime settings in global tcpip_settings structure

7.3.1 void (*tcpip_settings.serial_statuschange)(Uint16 state)

Callback function, called whenever a major status change occurs on the lower driver (currently only supported by the serial driver).

The state passed to the callback function can be:

LINESTATE_IDLE	linestate went idle (on disconnect and once on setup)
LINESTATE_CALLIN	incoming modem connection has been established
LINESTATE_CALLOUT	outgoing modem connection has been established
LINESTATE_CALLIN_PPPCONNECTED	incoming PPP connection has been established

LINESTATE_CALLOUT_PPPCONNECTED	outgoing PPP connection has been established

If the connection goes IDLE the callback function should at least:

- Reset the tcpip_settings.self_ip number if it was assigned by an outgoing PPP connection
- Reset any pending open tcpip sessions by calling tcp_resetall()

See the provided templates and demos for elaborate examples.

7.3.2 Uint8 tcpip_settings.self_ip[4]

This IP-address will be suggested for our use while building a PPP connection.

7.3.3 Uint8 tcpip_settings.ppp_client_ip[4]

(Only needed when SERIAL_PPP is defined)

IP-address suggested to other side while building a PPP connection.

7.3.4 Uint8 (*tcpip_settings.ppp_usercheck)(char *user, char *password)

(Only needed when SERIAL_PPP and SERIAL_CALLIN is defined)

Callback function used by PAP handshaking to verify if a user/password combination is allowed access.

parameter	**usage**
user	pointer to username string received by PAP
password	pointer to password string received by PAP
(returns)	<>0 when the combination is OK, 0 otherwise

7.3.5 char *tcpip_settings.ppp_out_user

(Only needed when SERIAL_PPP and SERIAL_CALLOUT is defined)

Must point to string with username to send to the other side for PAP authenticating.

7.3.6 char *tcpip_settings.ppp_out_password

(Only needed when SERIAL_PPP and SERIAL_CALLOUT is defined)

Must point to password string to send to the other side for PAP authenticating.

7.3.7 char *tcpip_settings.modem_init

Must point to initialisation string for modem, which will be send once to the modem upon startup of the stack.

7.3.8 char *tcpip_settings.modem_dialout_number

(Only needed when SERIAL_MODEM, SERIAL_PPP and SERIAL_CALLOUT is defined)

Must point to string with telephone number if a dialout connection is to be made.

7.4 Usage

7.4.1 Lowlevel init

With most drivers a lowlevel init call is needed. Implementation and calling syntax is platform dependant.

7.4.2 void modem_dialout_ppp(void)

Start a dialup and build a PPP connection. (If SERIAL_MODEM is not defined use ppp_logon() instead)

7.4.3 void ppp_logon(void)

Start a PPP connection on an open line. If a modem connection must be made beforehand call modem_dialout_ppp() instead (which should in fact always be done whenever SERIAL_MODEM is defined, unless you want to switch an open connection from a proprietary protocol to PPP).

7.4.4 void ppp_terminate(void)

Bring down a PPP-connection; if SERIAL_MODEM is defined the modem connection will be broken afterwards.

7.4.5 void modem_break(void)

Break a connected modem. Use only on SLIP connections, on PPP connections use ppp_terminate().

8 API - Ethernet transport

8.1 Source

common\ethernet.c common\ethernet.h	All ethernet/ARP code
common\drivers\eth_driver.h	Lowlevel driver header, mostly prototypes and defaults for platform dependant code
<platform>\serial_<platform>.c <platform>\serial_<platform>.h	Lowlevel driver code (platform dependant)

8.2 Compile time settings in tcpipset.h

8.2.1 #define OPTION_ETH

If defined this turns on the ethernet interface.

8.2.2 #define ARPCACHE_ENTRIES <entries>

(defaults to 8)

This is the number of IP/MAC combinations the driver keeps in memory. There is no concept of lifetime, but the least recently used entry is discarded first.

An IP/MAC translation is stored when receiving a IP packet or a repy to an ARP request. When sending a frame initiated from the stack it is technically *not* possible - due to the single-buffer no-copy design - for the stack to resolve this address on the fly using ARP.

If the IP-address is not found in the ARP cache, a frame to be send will be silently discarded. The easiest way around this limitation is to verify the IP number is still available in the cache before each triggering of a send from a TCP client or server.

Note: TCP clients or servers answering to incoming data do not have this problem, the IP/MAC translation needed has just been stored or refreshed upon reception of the incoming data IP-frame.

8.3 Runtime settings in global tcpip_settings structure

8.3.1 Uint8 tcpip_settings.mac[6]

This contains the MAC address of the ethernet adapter.

Depending on the lowlevel driver used, this can be (a) written to the hardware, (b) read from the hardware, (c) ignored.

8.4 Usage

8.4.1 General

Due to the use of only a single global buffer by the stack, it must already know which MAC-address belongs to which IP-address before a IP packet can be send. This follows from the fact that the single buffer is used for sending an ARP request, receiving the ARP answer and sending the IP packet.

The sequence to create a session to an IP-address should thus be:

```
If arp_resolve(ip-address) == NULL
      Call arp_send_request(ip-adress)
      while (arp_resolve(ip-address) <> NULL)
            wait a while

Call tcp_create() to set up a new session.
```

If the IP->MAC translation is not found in the ARP-cache while a IP-frame has to be transmitted by the stack, the frame will be dropped.

8.4.2 Lowlevel init

With most drivers a lowlevel init call is needed. Implementation and calling syntax is platform dependant.

8.4.3 void arp_send_request(Uint8 *ip)

Broadcasts an ARP request to resolve the given IP address into a MAC address. The reply is processed without intervention and stored in the cache. To check if the answer came in arp_resolve(ip) should be used.

8.4.4 Uint8 *arp_resolve(Uint8 *ip)

Checks if the ARP cache contains a MAC for the IP address given. If the MAC-address is found the fucntion will return a pointer to it, otherwise it will return NULL.

9 API - DHCP client

9.1 Source

common\dhcp.c common\dhcp.h	All DHCP code

9.2 Compile time settings in tcpipset.h

9.2.1 #define OPTION_DHCP

If defined a DHCP client is present and an IP address is requested from a DHCP server. If OPTION_DNS_CLIENT is defined a DNS IP address is taken from the server as well.

Only useful if OPTION_ETH is defined.

9.2.2 #define DHCP_TIMERTICK_MANUAL

For DHCP lease renewal it is important that time between calls to tcpipstack_timertick() is on average not longer than one second. If in doubt the DHCP_RENEWSLACK_SECS could be made a significant part of the DHCP_LEASETIME, to be sure a lease is renewed in time.

The alternative is to update the timer outside the stack under interrupt. If DHCP_TIMERTICK_MANUAL is defined, the timer is made volatile and automatic countdown on tcpipstack_timertick() is turned off (see also tcpip_setting.dhcp_timer).

9.2.3 #define DHCP_WAITOFFER_SECS <secs>

(default 10)

Time the stack will wait on an offer by a DHCP server before resending a discover.

9.2.4 #define DHCP_WAITACK_SECS <secs>

(default 10)

Time the stack will wait for an acknowledge by a DHCP server before resending a discover.

9.2.5 #define DHCP_RENEWSLACK_SECS <secs>

(default 10)

Time before the expiration of a lease the stack will send as renewal request.

9.2.6 #define DHCP_LEASETIME_SECS <secs>

(default 600)

Leasetime suggested to server when requesting or renewing a lease.

9.3 Runtime settings in global tcpip_settings structure

9.3.1 Uint8 tcpip_settings.dhcp_server_ip[4]

Filled by the module with the IP address of the DHCP server who gave us the current lease.

9.3.2 Uint8 tcpip_settings.self_ip[4]

Filled by the module with our IP address we can use for the duration of the lease.

9.3.3 Uint8 tcpip_settings.dns_world_ip[4].

Filled by the module with the IP addres of an outside DNS server we can use for the duration of the lease.

9.3.4 Uint16 tcpip_settings.dhcp_timer

Countdown timer for DHCP time-outs and lease expiration.

Only to be set by the programmer if DHCP_TIMERTICK_MANUAL is defined. In that case the variable will be volatile. Every second the following should be executed:

```
if (tcpip_settings.dhcp_timer)
{
        --tcpip_settings.dhcp_timer == 0;
}
```

(See also #define DHCP_TIMERTICK_MANUAL)

9.4 Usage

9.4.1 UDP_SERVERFUNCTION dhcp_client

UDP server to process for incoming DHCP answers, must be included in the tcpip_settings.udpclientserver[] list on port UDP_PORT_DHCP_CLIENT.

9.4.2 Automatic behaviour

No user-written code is needed (besides the optional manual timer descibed above), upon init of the stack the first discover is broadcast. The module will automatically process incoming replies, renew the lease when it is about to expire, and resend a discover if the original server doesn't respond anymore.

10 API - DNS server & client

10.1 Source

common\dns.c common\dns.h	All DNS server & DNS client code

10.2 Compile time settings in tcpipset.h

10.2.1 #define OPTION_DNS_SERVER

If defined a DNS server will be compiled. The only request the server can process is to convert the name of the device into its IP-address.

OPTION_UDP has to be active as the DNS is UDP based.

10.2.2 #define OPTION_DNS_CLIENT

If defined a DNS client will be compiled. OPTION_UDP has to be active as the DNS is UDP based.

10.3 Runtime settings in global tcpip_settings structure

10.3.1 char *tcpip_settings.dns_self_name

(only used when OPTION_DNS_SERVER is set)

The hostname under which our mini-DNS-server knows us.

10.3.2 Uint8 tcpip_settings.self_ip[4]

The IP-address under which our mini-DNS-server knows us.

10.3.3 Uint8 tcpip_settings.dns_world_ip[4]

(only used when OPTION_DNS_CLIENT is set)

The IP address of the DNS server in the world we know about. Can be filled by ourselves, by PPP or by DHCP.

10.3.4 void tcpip_settings.(*dns_portip)(Uint16 port, char *ip)

(only used when OPTION_DNS_CLIENT is set)

User-supplied callback function used by the DNS client if a resolved IP number answer comes in on a UDP port. The user must keep track which request was send using which UDP port.

API for the callback function:

parameter	**usage**
port	UDP port on which the answer came in. This way the same callback function can be used to process multiple outstanding requests.

ip	IP-address the DNS server told belonged to the IP-address we asked about

10.4 Usage

10.4.1 UDP_SERVERFUNCTION dns_server

(only used when OPTION_DNS_SERVER is set)

Mini-DNS-server for incoming DNS requests, must be include in the tcpip_settings.udpclientserver[] list on port UDP_PORT_DNS_SERVER.

The server has severe limited functionality: it can only resolve our hostname into our IP-address. All other requests are ignored.

10.4.2 UDP_SERVERFUNCTION dns_client

(only used when OPTION_DNS_CLIENT is set)

Server for incoming DNS answers, must be include in the tcpip_settings.udpclientserver[] list on the port given by dns_create().

10.4.3 Uint16 dns_create(hostname)

(only used when OPTION_DNS_CLIENT is set)

Send a DNS request to resolve a name. The client-side portnumber used for the request will be returned. A dns_client serverfunction should be activated on the port returned (insert the client in the udpclientserver[] list).

When the answer comes in the dns_client serverfunction will use the callback function dns_portip(dnsport, ip_found) to report back the answer to the usercode.

10.4.4 Automatic behaviour

Incoming DNS requests for resolving our name into an IP address will automatically be processed by the server if OPTION_DNS_SERVER is defined. All other requests will be ignored

11 API - FTP server

11.1 Source

common/servers/ftp_server.c common/servers/ftp_server.h	All FTP server code

11.2 Compile time settings in tcpipset.h

11.2.1 #define OPTION_FTP_SERVER

If defined an FTP server will be compiled.

11.2.2 #define FTP_VERBOSE

If defined, the status returned to the client will have a normal English description. If not defined, the description will be kept to a minimum (mostly "ok" or "err", usable for experienced users or automated clients). Verbosity costs about 500 bytes in constant strings.

11.2.3 #define FTP_SESSIONS <count>

Maximum number of simultaneous FTP sessions, maximum number useful would be TCP_SESSIONS. The RAM cost is one FTPSESSION structure per session.

11.2.4 #define FTP_SUBDIR_MAXDEPTH <levels>

Maximum levels of nested subdirectories supported by FTP (the filesystem itself does not have such a limitation). The level includes any number used for the root directory of the user.

11.3 Runtime settings in global tcpip_settings structure

11.3.1 Uint8 tcpip_settings.ftp_server_usercheck(char *user, char*password, char *rootdir)

Callback function used by the FTP server to verify user/password combinations. If wanted a root directory for the given user can be written to rootdir, default will be "/home/<user>".

parameter	usage
user	Pointer to username string received by FTP
password	Pointer to password string received by FTP
rootdir	Optionally a virtual root-directory string can be written to this pointer. If not its default will be /home/xxxx, where xxxx is the user name Example: if access to the full 'disk' must be had fill rootdir with "/".
(returns)	Return <>0 when the combination is OK, 0 otherwise

11.4 Usage

11.4.1 TCP_SERVERFUNCTION ftp_server

Server for incoming FTP requests, must be include in the tcpip_settings.tcpservers[] list on port TCP_PORT_FTP_SERVER.

11.4.2 Automatic behaviour

Incoming FTP requests will automatically be processed by the server.

12 API - HTTP server

12.1 Source

common/servers/http_server.c common/servers/http_server.h	All HTTP server code

12.2 Introduction

The HTTP server can be used for three major functions or combinations of them:

12.2.1 Serving hardcoded pages

A requested url is compared with a hardcoded list or urls. If a match is found a related hardcoded page is returned.

12.2.2 Serving from a live filesystem

A requested url is searched as filename on the filesystem, optionally starting from a virtual rootdirectory. If a match found the file is returned.

12.2.3 Interfacing to the embedded device using CGI-variables

This is at one hand a separate dynamic interface, and on the other hand an elaboration on the serving of hardcoded pages. See below for more details.

12.3 CGI-variables

The HTTP server knows the concept of 'CGI-variables'. It can have a list of multiple CGI-variable definitions. The easiest way to set up such is list is by using the supplies GUI tool.

For the server a variable has a 'uservisible name string' and a 'uservisible value string', but every variable has as part of its definition a conversion-function-callback (called 'CGI-handler') which links it to an embedded counterpart. For some simple types (for instance C-unsigned-short) handler-functions are included, other ones, maybe interfacing directly with some hardware, must be user-written.

The variables can be used in several ways:

12.3.1 SSI-tags

In hardcoded pages special tags can be replaced with the value of a CGI-variable.

12.3.2 POST-method

On specific urls (default /cgi/varname) the value of a CGI-variable can be read or written.

12.3.3 INFO interface

On a specific url (default /cgi/info) an overview of the available variables and their acces-rights can be read into a browse. The format used is human-readable but suitable for automated processing.

12.4 Using SSI

Server Side Includes (SSI) are only supported on hardcoded HTML pages (meaning *not* from the live filesystem).

The HTML syntax to including the value of a variable (or to be more precise: the output of its handler function called with cgimode=CGIMODE_SSIGET) is:

```
<!--#echo var="varname"-->
```

Where "varname" is the quoted external name of the variable wanted.

12.5 Generated example

When using the tools to generate the arrays with hardcoded files, a sample form will be generated which shows the SSI-call for all variables where it was defined available, and code to call the correct URLs for setting values for all variables who were defined writable.

12.6 Compile time settings in tcpipset.h

This server needs a complete file file of settings to define its behaviour and to describe the website it serves. In tcpipset.h only a few settings are be present, of which HTTP_DEF is the most important one as it tells the server where to find the other settings.

12.6.1 #define OPTION_HTTP_SERVER

If defined an HTTP server will be compiled.

12.6.2 #define HTTP_DEF <headerstring>

This should contain name of the header with the HTTP server definition. This file should define all website aspects of the server.

12.6.3 #define HTTP_DIRINDEX <indexnamestring>

If defined the server serves this file from a directory if the directory itself is requested.

12.6.4 #define HTTP_FS_ROOT <virtualrootstring>

If defined the HTTP server tries to find urls asked on the filesystem, using this path as virtual root.

12.6.5 #define HTTP_SSIGETVAR_MAXLEN <chars>

Maximum length of the usernames used by SSI-accessible variables.

12.7 Compile time settings in file included with HTTP_DEF

Normally this file is generated by the configuration tools. The information given below is very concise and probably scarcely sufficient to build a complete server definition from scratch. If you need a special configuration impossible to generate, start with one generated as close to your goal as possible and work from there.

12.7.1 #define HTTP_MIMETYPES <mimetypearray>

Links url extensions to mimetypes, must be an array like:

```
{
        // extension, mimetype
        { "html", "text/html},
        ...etc...,
        { 0, NULL}
}
```

If the server finds no match for an url, the first entry is used.

12.7.2 #define HTTP_ERRORPAGE_* <errorstring>

If defined these override the default content of the generated error pages:

HTTP_ERRORPAGE_NOTFOUND	default "<H2>HTTP 404 File not found</H2>"
HTTP_ERRORPAGE_INVALIDCALL	default "<H2>HTTP 490 Invalid CGI call</H2>"
HTTP_ERRORPAGE_INVALIDVALUE	default "<H2>HTTP 491 Invalid CGI value</H2>"

12.7.3 #define HTTP_PAGES <pagearray>

If defined this contains all hardcoded pages on fixed URLs, it should be an array like:

```
{
        // url, page, pagelength
        { "/index.html", <string with page>, <pagelength>},
        { "/logo.gif", <pointer to binary logo>, <size of logo>},
        ...etc...,
        { 0, NULL, 0}
}
```

12.7.4 #define HTTP_CGIURL_* <urlstring>

If defined these override the default URL locations of the CGI interface:

HTTP_CGIURL_GET	default "/cgi/get"
HTTP_CGIURL_SET	default "/cgi/set"
HTTP_CGIURL_INFO	default "/cgi/info"

12.7.5 #define HTTP_CGIVAR_*

These defines turn on the CGI interface code in the server for a specific type. These are only a few predefined interface functions to get started, most projects will probably implement their own specific CGI handlers.

HTTP_CGIVAR_WORD	16 bit unsigned integer with min/max value
HTTP_CGIVAR_STRING	string array with min/max length
HTTP_CGIVAR_BOOL	boolean with on/off string
HTTP_CGIVAR_IP	xxx.xxx.xxx.xxx formatted IP address (read only, not write)

12.7.6 #undef HTTP_CGI_*

If one of these is ***undefined*** the code for that type of access is removed from the server:

HTTP_CGI_GET	Variable is readable with cgi interface
HTTP_CGI_SSISET	Variable is readable with a SSI tag in a hardcoded HTML page
HTTP_CGI_SET	Variable is writable with cgi interface
HTTP_CGI_INFOGET	Variable is included in cgi interface description

12.7.7 #define HTTP_CGIVARS <cgidefinitionarray>

If defined this contains a list of C-variables to be publicized with the default CGI-interface, should be an array like:

{

{<HTTP_CGI*-flags>, <pointer to var>, <human-name>, <v1>, <v2>, <pointer to handler>},

... etc...,

{0, NULL, NULL, NULL, NULL, NULL}

}

The pre-defined handlers, v1/v2 meanings, and the HTTP_CGIVAR_* defines that turn on the handler code are:

cgifunc_word	v1/v2=min/max value	HTTP_CGIVAR_WORD
cgifunc_string	v1/v2=min/max length	HTTP_CGIVAR_STRING
cgifunc_bool	v1/v2=off/on human-text	HTTP_CGIVAR_BOOL
cgifunc_ip (only reading, not writing)	v1/v2=not used	HTTP_CGIVAR_IP

12.8 CGI handler functions

Handler functions for accessing 16-bit, 8-bit values and boolean values are included. Of course you can have other requirements: for instance the value you need is not located in normal memory but has to be read from certain registers, or if you want to process the incoming data instead of just writing it to memory. In these cases you have to write your own handler.

The API for a handler is:

```
CALLBACKMEMSPEC Uint8 foo(Uint8 cgimode, CGIVAR ROMMEMSPEC * var, char *url)
```

With most platforms CALLBACKMEMSPEC will be empty, and ROMMEMSPEC will just be 'const', but they are predefined by the stack to values appropriate for your platform.

The cgimode tells what action the function should carry out:

CGIMODE_SET	HTTP received "<cgiurl>?varname=value", url points to incoming 'value' (in the URL string)
CGIMODE_SETRESULT	write (in the buffer) your answer to the previous CGIMODE_SET
CGIMODE_SSIGET	write (in the buffer) the value for a SSI tag with the current var
CGIMODE_GET	HTTP received "<cgiurl>?varname", write (in the buffer) your answer

CGIMODE_INFOGET	write in the buffer a single line description of your interface

If any action is not supported or an illegal value is received the function should return 0, otherwise 1.

The CGIVAR is a structure with the following elements relating to the variable for which the fucntion was called (all values are normally filled in using the GUI setup tool while you declare the HTTP CGI interface):

void *var	pointer to the variable (NULL if function doesn't refer to a variable)
char ROMMEMSPEC *name	pointer to the name the HTTP server (and thus the user) knows
void ROMMEMSPEC *info1	userdefinable, for instance usable as a minimum allowable value
void ROMMEMSPEC *info2	userdefinable, for instance usable as a maximum allowable value

Important: writing back an answer must always be done using the http_bufwrite_*() functions exported by the server. For examples how to use these see the demo sources.

12.9 Usage

12.9.1 http_server()

Server for incoming HTTP requests, must be include in the tcpip_settings.tcpservers[] list on port TCP_PORT_HTTP_SERVER.

12.9.2 Automatic behaviour

Incoming HTTP requests will automatically be processed by the server.

13 API - TELNET server

13.1 Source

common/server/telnet_server.c common/server/telnet_server.h	All TELNET server code

13.2 Compile time settings in tcpipset.h

13.2.1 #define OPTION_TELNET_SERVER

If defined a TELNET server will be compiled. The server does nothing useful without modifications, all commands recognized beyond "help" and "quit" must be implemented by the developer.

13.2.2 #define TELNET_SESSIONS

(default 2)

Maximum number of simultaneous TELNET sessions, maximum number useful would be TCP_SESSIONS. The RAM cost is one TELNETSESSION structure per session, were the size of the commandbuffer (definable by TELNET_BUFLENGTH) is the most important issue.

13.2.3 #define TELNET_BUFLENGTH

(default 12)

Maximum length of commandline processed, any more characters entered afterwards are discarded. The default is usable for the template and demos, a production commandprocessor usually needs a bigger buffer.

13.3 Runtime settings in global tcpip_settings structure

13.3.1 tcpip_settings.(*telnet_server_usercheck) (char *user, char *password, **commandprocessor(...))

Callback function used by the TELNET server to verify user/password combinations. The callbackfunction must fill in the commandprocessor to be used for the user in question.

13.4 Command processors

A commandprocessor is a callback function which processes the commands entered by a user. A single command procesor may be used for all sessions, or a different one may be selected by the user/password verifying function, based on username or whatever criteria is relevant to the application.

The API for a commandprocessor is:

```
Uint8 commandprocessor(char *cmd, char *buf, char *user, Uint8 userid)
```

The processor receives a command, a buffer were to place its reply, and a pointer to the username and userid. As soon as the function returns zero the user is logged out of telnet.

An example comandprocessor can be found in the template directory and is used in the demo.

13.5 Usage

13.5.1 telnet_server()

Server for incoming TELNET requests, must be include in the tcpip_settings.tcpservers[] list on port TCP_PORT_TELNET_SERVER.

13.5.2 Automatic behaviour

Incoming TELNET requests will automatically be processed by the server.

14 API - SMTP client

14.1 Source

common/server/smtp_client.c common/server/smtp_client.h	All SMTP client code

14.2 Compile time settings in tcpipset.h

14.2.1 #define OPTION_SMTP_CLIENT

If defined an SMTP client will be compiled. The client can process a script (array of expected server codes and strings to reply, can be build dynamically).

OPTION_UDP has to be defined.

14.3 Usage

14.3.1 smtp_client()

Client for incoming TELNET requests. Must be passed when creating a TCP session to a server, like:

```
tcpsession = tcp_create(0, smtp_server_ip, TCP_PORT_SMTP, &smtp_client)
```

If the tcpsession <> NULL, the client can be initialized to an SMTP script with

```
tcpsession.cargo.smtp_clinet.nextmsg = <pointer to SMTP_MSG[] scriptline>
```

A script is a SMTP_MSG array like:

```
SMTP_MSG my_script[] =
{
       {<status expected from server>, <line to send>}
       ... etc...,
       {NULL, NULL}
}
```

So for a minimal message to a normal SMTP server:

```
SMTP_MSG my_script[] =
{
    {"2", "HELO smtpmailserver.somedomain.example"},
    {"2", "MAIL From: <auto.generated@myserver.example>"},
    {"250", "RCPT To: <someone.else@someserver.example>"},
    {"2", "DATA"},
    {"3", "From: <auto.generated@myserver.example>"},
    {NULL, "Subject: Hello World"},
    {NULL, "This is the only message line."},
    {NULL, "."},
    {"2", "QUIT"},
    {NULL, NULL}
}
```

(For a complete example see the demo and template source.)

15 API - FAT-based Filesystem

15.1 source

common/filesystem/filesys.c common/filesystem/filesys.h	All highlevel filesystem code
common/filesystem/diskdrv.c common/filesystem/diskdrv.h	Lowlevel RAMdisk driver for the filesystem

The lowlevel RAMdisk driver expects the diskimg in a global Uint8 diskimg[FS_DISKSIZE].

15.2 Compile time settings in tcpipset.h

15.2.1 #define FS_DEF

This should contain name of the header with the filesystem definition. This file should define all aspects of the disk and filesystem.

15.3 Compile time settings in file included with FS_DEF

15.3.1 #define FS_READONLY

If defined the filesystem cannot be written to.

15.3.2 #define FS_DISKSIZE

Disksize in bytes.

15.3.3 #define FS_CLUSTERSIZE

Clustersize in bytes.

15.3.4 #define FS_ROOTENTRIES

Maximum number of entries in the root directory.

15.3.5 #define FS_MAXFILENAMELEN

Maximum filename length (excluding terminating \0).

15.4 Usage

15.4.1 FCB allocation

The filesystem does ***not*** allocate FCBs itself, the caller is responsible to allocate and free them. The FS_FILEHANDLE is just a pointer to such a caller-allocated structure.

All calls support a root, path and name, this can be used to support virtual roots and avoids allocating memory to temporarily concatenate filenames with paths and virtual roots.

15.4.2 int fs_create(FS_HANDLE handle, char * root, char * path, char * name)

Create a file for writing.

15.4.3 int fs_mkdir(char * root, char * path, char * name)

Create a directory.

15.4.4 int fs_open(FS_HANDLE handle, char * root, char * path, char * name)

Open a file.

15.4.5 void fs_close(FS_HANDLE handle)

Close an open file.

15.4.6 int fs_remove(char * root, char * path, char * name)

Remove a file.

15.4.7 void fs_format(char * vollabel)

Format the drive.

15.4.8 int fs_rmdir(char * root, char * path, char * name)

Remove a directory.

15.4.9 int fs_rename(char * root, char * path, char * oldname, char * newname)

Rename a file.

15.4.10 Uint32 fs_seek(FS_HANDLE handle, Sint32 offset, FS_SEEK whence)

Seek to a specified position in a file.

15.4.11 int fs_write(FS_HANDLE handle, void * buf, Sint16 size)

Write a block of data to the file.

15.4.12 fs_read(FS_HANDLE handle, void * buf, Sint16 size)

Read a block of data from the file.

15.4.13 int fs_findfirst(FS_HANDLE handle, char *root, char *path, FS_DIRENTRY *dir)

Find first entry of a directory.

15.4.14 int fs_findnext(FS_HANDLE handle, FS_DIRENTRY * dir)

Find next entry of the directory.

15.4.15 void fs_closedir(FS_HANDLE handle)

Close directory after searching.

15.4.16 Uint16 fs_stat(char *root, char *path, char * name)

Get file status.

16 Creating your own TCP client or server

16.1 Introduction

Included with the stack in the <platform>/demo/userclientserver directory you can find three related demo's. They consist of a server pretending to serve images from a camera, and two functional equal clients downloading those images.

Depending on the platform the communication used is SLIP or ethernet (for some platforms both transports are included as separate projects).

16.1.1 Protocol

Default the server runs on TCP port 4000, and uses the following protocol:

client request	server response	comment
R<width><height>	r<width><height>	server receives & returns size (Uint8 x Uint8) to be grabbed and transmitted
G	g	server grabs a new image from the camera
I	i<imagedata>	server transmits width x heigth databytes

The client will execute the following steps:

1. Create a session connected to the server at port 4000
2. Send a 'R' request to set the resolution
3. Wait for the 'r' answer to confirm the resolution
4. Send a 'G' request to let the server grab a new image
5. Wait for the 'g' answer to confirm the grab
6. Send an 'I' request to ask for the imagedata
7. Wait for the 'i' answer to confirm imagedata request
8. Receive the image data
9. Start over with step (4) to get another image

If the session is aborted at any time, the client will immediately try to reconnect.

16.1.2 Dumb or smart client

There are two versions of the client:

- Dumb client-function: The client-function can send specific requests and wait for their answers. The main program handles the procedure as described above. It constantly monitors the state of the client, and triggers it to send new requests when the answer to previous one has been received.
- Smart client-function: The client-function itself handles the complete procedure as described above. The main program only guards if the session is still alive or has to be rebuild.

There is no specific technical advantage to either method for this protocol. You can choose one based on convenience or readability in your situation.

In a situation where there is no incoming data and a client or server must initiate a transmission, you must trigger this from the main program. The client/server-function will only be called when any data is received or a resend is needed, thus it is not possible to initiate a new transmission while the other side if the communication is silent. In these cases triggering the client/server function from outside like implemented in the dumb client is required.

16.2 How to create a server

16.2.1 Create the server-function (my_server.c/h)

From templates/template_tcp.c/h my_server.c/h are copied. Using the comments in the template we fill in what we want the server to do:

- Reply on 'R', 'G' and 'I' commands as per defined protocol
- If the last command was 'I' and we have more image data to send, send the next chunk of image data
- Remember the last command processed, if we are asked to retry then recreate the original *incoming* frame, so the server will regenerate the same frame as the first time.

In the header a cargo-structure is defined, for which the stack will give us a unique copy belonging to every active session. In the cargo we keep track of the active width/height for the session, and we store the last command and transfer position to be able to recreate the last frame if resends are requested.

For any specific details of the implementation see my_server.c/h, and optionally compare it with the template_tcp.c/h it was based upon.

16.2.2 Configure the stack

With the stackconf tool a very basic configuration is generated, in fact only plain TCP with no servers defined except ICMP echo because that might be useful for initial tests.

All values are left on initial defaults except:

- If we use the serial transport it is set to SLIP, call-in only, no modem support
- If we use the ethernet transport DHCP is turned off (we will use a static IP address)
- UDP is turned off, all default servers turned off.

16.2.3 Create the main loop (main.c)

From templates/template_main.c a main.c is copied, with everything stripped not relating to SLIP or TCP (the #ifdef tests in the template and the generated defines in tcpipset.h show you what code is used).

The server-function is added on port 4000 to a TCP_SERVER array, which is added to the stacks tcpip_settings.

16.2.4 Create the project-specific stack settings (tcpipsysset.h)

From <platform>/template_tcpsysset.h a tcpipsysset.h is copied. We add the following lines

```
#define USER_CLIENTSERVER1_H            "my_server.h"
#define USER_CLIENTSERVER1_CARGOTYPE    MY_SERVER_CARGO
```

```
#define USER_CLIENTSERVER1_CARGO        my_server
```

to let the stack know about the server we created.

16.3 How to create a smart client

16.3.1 Create the client-function

From templates/template_tcp.c/h my_client.c/h are copied. Using the comments in the template we fill in what we want the client to do:

- Send 'R', 'G' and 'I' commands as per defined protocol, wait on their responses
- Receive the responses to the 'R', 'G' and 'I' commands as per defined protocol, if response to 'I' is received wait for the image data
- Receive the image data
- Implement the steps needed to get images in a continuous loop (Send 'R', receive 'r', send 'G', receive 'g', send 'I', receive 'i' and image data, etc., etc.)
- Remember the last command processed, if we are asked to retry then recreate the original *incoming* frame, so the server will regenerate the same frame as the first time.

In the header a cargo-structure is defined, which the stack will give us a unique copy for every active session. In the cargo we keep track of the active width/height for the session, and we store the last command and transfer position to be able to recreate the last frame if resends are requested. Also we keep a state-machine variable to execute all steps in order.

For any specific details of the implementation see my_server.c/h, and optionally compare it with the template_tcp.c/h it was based upon.

16.3.2 Configure the stack

The configuration used is identical to that of the server, with on exception: if SLIP is used call-out is selected instead of call-in.

16.3.3 Create the main loop (main.c)

From templates/template_main.c a main.c is copied, with everything stripped not relating to SLIP or TCP (the #ifdef tests in the template and the generated defines in tcpipset.h show you what code is used).

The TCP_SERVER array is left empty as we have no servers defined but are only going to use our function as a client.

If no session is active we open a session to port 4000, we keep monitoring this session to see if it is still active. If not, we try to open a new session.

16.3.4 Create the project-specific stack settings (tcpipsysset.h)

From <platform>/template_tcpsysset.h a tcpipsysset.h is copied. We add the following lines

```
#define USER_CLIENTSERVER1_H            "my_smartclient.h"
#define USER_CLIENTSERVER1_CARGOTYPE    MY_CLIENT_CARGO
```

```
#define USER_CLIENTSERVER1_CARGO        my_client
```

to let the stack know about the client we created.

16.4 How to create a dumb client

16.4.1 Create the client-function

From templates/template_tcp.c/h my_client.c/h are copied. Using the comments in the template we fill in what we want the client to do:

- If externally triggered to do so, send 'R', 'G' and 'I' commands as per defined protocol, wait on their responses
- Receive the responses to the 'R', 'G' and 'I' commands as per defined protocol, if response to 'I' is received wait for the image data
- Receive the image data
- Remember the complete last frame send, if we are asked to retry then just resend the previous frame (this is different from the smart client to demonstrate another method to implement retries).

In the header a cargo-structure is defined, which the stack will give us a unique copy for every active session. In the cargo we keep track of the active width/height for the session, of the state we are in, and we store a copy of the last frame transmitted.

For any specific details of the implementation see my_server.c/h, and optionally compare it with the template_tcp.c/h it was based upon.

16.4.2 Configure the stack

The configuration used is identical to that of the smart client.

16.4.3 Create the main loop (main.c)

From templates/template_main.c a main.c is copied, with everything stripped not relating to SLIP or TCP (the #ifdef tests in the template and the generated defines in tcpipset.h show you what code is used).

The TCP_SERVER array is left empty as we have no servers defined but are only going to use our function as a client.

If no session is active we open a session to port 4000, we keep monitoring this session to see if it is still active. If not, we try to open a new session.

16.4.4 Create the project-specific stack settings (tcpipsysset.h)

From <platform>/template_tcpsysset.h a tcpipsysset.h is copied. We add the following lines

```
#define USER_CLIENTSERVER1_H            "my_dumbclient.h"
#define USER_CLIENTSERVER1_CARGOTYPE    MY_CLIENT_CARGO
#define USER_CLIENTSERVER1_CARGO        my_client
```

to let the stack know about the client we created.

17 Bibliography

1. Postel, J.: *User Datagram Protocol*, RFC-768, Information Sciences Institute, August 1980
2. Postel, J., ed.: *Internet Protocol - DARPA Internet Program Protocol Specification*, RFC-791, Information Sciences, Institute, September 1981
3. Postel, J.: *Internet Control Message Protocol - DARPA Internet Program Protocol Specification*, RFC-792, Information Sciences Institute, September 1981
4. Postel, J., ed.: *Transmission Control Protocol - DARPA Internet Program Protocol Specification*, RFC-793, Information Sciences Institute, September 1981
5. Plummer, D.: *An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48-bit Ethernet Addresses for Transmission on Ethernet Hardware*, RFC-826, MIT-LCS, November 1982
6. Postel, J. Reynolds: *File Transfer Protocol (FTP)*, RFC-959, Information Sciences Institute, October 1985
7. Mockapetris, P.: *Domain Names - Concepts and Facilities*, RFC-1034, Information Sciences Institute, November 1987
8. Mockapetris, P.: *Domain Names - Implementation and Specification*, RFC-1035, Information Sciences Institute, November 1987
9. G. McGregor: *The PPP Internet Protocol Control Protocol (IPCP)*, RFC-1332, Merit, May 1992
10. B. Lloyd, L&A, W. Simpson: *PPP Authentication Protocols*, RFC-1334, Daydreamer, October 1992
11. Reynolds, J. Postel: *Assigned Numbers*, RFC-1340, Information Sciences Institute, July 1992
12. Simpson, W., ed.: *The Point-to-Point Protocol (PPP)*, RFC-1661, Daydreamer, July 1994
13. Simpson, W., ed.: *PPP in HDLC-like Framing*, RFC-1662, Daydreamer, July 1994
14. S. Cobb: *PPP Internet Protocol Control Protocol Extensions for Name Server Addresses*, RFC-1877, Microsoft, December 1995
15. K. Schneider, S. Venters: *PPP Serial Data Transport Protocol (SDTP)*, RFC-1963, ADTRAN, Inc., August 1996
16. R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee: *Hypertext Transfer Protocol -- HTTP/1.1*, RFC-2068, DEC MIT/LCS, January 1997
17. R. Droms: *Dynamic Host Configuration Protocol*, RFC-2131, Bucknell University, March 1997
18. S. Alexander: *DHCP Options and BOOTP Vendor Extensions*, RFC-2132, Bucknell University, March 1997