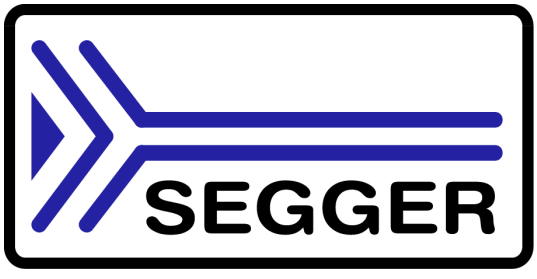
embOS/IP

CPU independant
TCP/IP stack for
embedded applications

User & Reference Guide

Document: UM07001
Software version: 2.10
Revision: 0
Date: September 13, 2012



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH & Co. KG (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel.+49 2103-2878-0
Fax.+49 2103-2878-28
E-mail: support@segger.com
Internet: http://www.segger.com

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: September 13, 2012

Software	Revision	Date	By	Description
2.10	0	120913	OO	Minor updates and corrections. Chapter "UPnP (Add-on)" added. Chapter "VLAN" added. Chapter "Core functions" updated. * IP_NI_ForceCaps() added. * IP_ARP_ConfigAgeout() added. * IP_ARP_ConfigAgeoutNoReply() added. * IP_ARP_ConfigAgeoutSniff() added. * IP_ARP_ConfigAllowGratuitousARP() added. * IP_ARP_ConfigMaxRetries() added. * IP_ARP_ConfigNumEntries() added. * IP_IFaceIsReadyEx() added. * IP_IGMP_Add() added. * IP_IGMP_JoinGroup() added. * IP_IGMP_LeaveGroup() added. Chapter "UDP zero-copy interface" updated. * IP_UDP_GetFPort() added. Chapter "Web server (Add-on)" updated. * Information regarding file uploads added. * More detailed description about multiuple connections added. * IP_WEBS_AddFileTypeHook() added. * IP_WEBS_AddVFileHook() added. * IP_WEBS_ConfigSendVFileHeader() added. * IP_WEBS_ConfigSendVFileHookHeader() added. * IP_WEBS_GetParaValuePtr() added. * IP_WEBS_SendHeader() added. Chapter "PPP/PPPoE (Add-on)" updated. * IP_MODEM_Connect() added. * IP_MODEM_Disconnect() added. * IP_MODEM_GetResponse() added. * IP_MODEM_SendString() added. * IP_MODEM_SendStringEx() added. * IP_MODEM_SetAuthInfo() added. * IP_MODEM_SetConnectTimeout() added. * IP_MODEM_SetInitCallback() added. * IP_MODEM_SetInitString() added. * IP_MODEM_SetSwitchToCmdDelay() added.
2.02c	0	120706	OO	Minor updates and corrections.
2.02a	0	120514	OO	Chapter "AutoIP" added. Chapter "Address Collision Detection" added.
2.02	0	120507	OO	Documentation updated for embOS/IP V2 stack. Chapter "API functions" updated. * "IP_GetRawPacketInfo()" added. * "IP_ICMP_Add()" added. * "IP_TCP_Add()" added. * "IP_UDP_Add()" added. Chapter "PPP" added. Chapter "NetBIOS" added.
1.60	0	100324	SK	Chapter "API functions" updated. * "IP_SetSupportedDuplexModes()" added. Chapter "FTP client" added. Minor updates and corrections.
1.58	0	100204	SK	Chapter "SMTP client" updated. Chapter "Configuration" updated. * Section "Required buffers" updated. Minor updates and corrections.
1.56	0	090710	SK	Chapter "API functions" updated. * "IP_DNSC_SetMaxTLL()" added. Chapter "Configuring embOS/IP" updated. * Macro "IP_TCP_ACCEPT_CHECKSUM_FFFF" added.

Software	Revision	Date	By	Description
1.54b	0	090603	SK	Chapter "Web server (Add-on)" updated. * "IP_WEBS_Process()" updated. * "IP_WEBS_ProcessLast()" added. * "IP_WEBS_OnConnectionLimit()" updated.
1.54a	1	090520	SK	Chapter "API functions" updated. * IP_GetAddrMask() updated. * IP_GetGWMask() updated. * IP_GetIPMask() updated. Chapter "Web server (Add-on)" updated. * Section "Changing the file system type" added. * Section "IP_WEBS_SetFileInfoCallback" updated.
1.54a	0	090508	SK	Chapter "Web server (Add-on)" updated. * IP_WEBS_GetNumParas() added. * IP_WEBS_GetParaValue() added. * IP_WEBS_DecodeAndCopyStr() added. * IP_WEBS_DecodeString() added. * IP_WEBS_SetFileInfoCallback() added. * IP_WEBS_CompareFilenameExt() added. * Section "Dynamic content" added * Section "Common Gateway interface" moved into section "Dynamic content". Chapter "Socket interface" * getpeername() corrected. Chapter "Network interface drivers" updated.
1.54	0	090504	SK	Chapter "UDP zero-copy" updated.
1.52	1	090402	SK	Chapter "SMTP client" added.
1.52	0	090223	SK	Chapter "API functions": * IP_SetTxBufferSize() added. * IP_GetIPAddr() updated. * IP_PrintIPAddr() updated.
1.50	0	081210	SK	Chapter "API functions": * IP_ICMP_SetRxHook() added. * IP_SetRxHook() added. * IP_SOCKET_SetDefaultOptions() added. * IP_SOCKET_SetLimit() added.
1.42	0	080821	SK	Chapter "Web server (Add-on)": * List of valid values for CGI parameter and values added. Chapter "FTP Server (Add-on)": * Section "FTP server system time" added. * pfGetTimeDate() added.
1.40	0	080731	SK	Chapter "API functions": * IP_TCP_SetConnKeepaliveOpt() added. * IP_TCP_SetRetransDelayRange() added. * IP_SendPacket() added. Chapter "Socket interface": * getsockopt() updated. * setsockopt() updated. Chapter "OS integration": * IP_OS_WaitItemTimed() added.
1.30	1	080610	SK	Chapter "FTP server (Add-on)" section "Resource usage" added Chapter "Web server (Add-on)" section "Resource usage" added
1.30	0	080423	SK	Chapter "FTP server (Add-on)" added. Chapter "Web server (Add-on)" updated.
1.24	3	080320	SK	Chapter "Socket interface": * getpeername added. * getsockname added.
1.24	2	080222	SK	Chapter "Device Driver": * NXP LPC23xx/24xx driver added.
1.24	1	080124	SK	Chapter "HTTP server (Add-on)" updated. Chapter "API functions": * IP_UTIL_EncodeBase64() added. * IP_UTIL_DecodeBase64() added.

Software	Revision	Date	By	Description
1.24	0	080124	SK	Chapter "HTTP server (Add-on)" added: Chapter "API functions": * IP_AllowBackPressure() added. * IP_GetIPAddr() added. * IP_SendPing() added. * IP_SetDefaultTTL() added.
1.22	4	071213	SK	Chapter "Introduction": * Section "Components of an Ethernet system" added. Chapter "API functions": * IP_IsIFaceReady() added. * IP_NI_ConfigPHYAddr() added. * IP_NI_ConfigPHYMode() added. * IP_NI_ConfigBasePtr () added. Chapter "Socket interface": * All functions: parameter description enhanced. Chapter "Device drivers" renamed to "Network interface drivers". Chapter "Network interface drivers": * Section "ATMEL AT91SAM7X" added. * Section "ATMEL AT91SAM9260"added. * Section "Davicom DM9000"added. * Section "ST STR912"added.
1.22	3	071126	SK	Chapter "OS Integration": * IP_OS_Sleep() removed. * IP_OS_Wakeup() removed. * IP_OS_WaitItem added. * IP_OS_SignalItem added. Chapter "Running embOS/IP on target hardware" updated.
1.22	2	071123	SK	Chapter "Socket interface": * gethostbyname() added. * Structure hostent added. Chapter "Core functions": * IP_PrintIPAddr() added. * IP_DNS_SetServer() added.
1.22	1	071122	SK	Chapter "DHCP": * IP_DHCPC_Activate() updated. Chapter "Debugging": * Section "Testing stability" added. Chapter "Socket interface": * Section "Error codes" added.
1.22	0	071114	SK	Chapter "Introduction": * "Request for comments" enhanced. Chapter "API functions": * IP_AddLogFilter() added. * IP_AddWarnFilter() added. * IP_GetCurrentLinkSpeed() added. * IP_TCP_Set2MSLDelay() added. * select() added. Various function descriptions enhanced. Chapter "API functions" renamed to "core functions". Socket functions removed from chapter "API functions" Chapter "Socket interface" added. Chapter "DHCP" added. Chapter "UDP zero copy" added. Chapter "TCP zero copy" added. Chapter "Glossary" added. Chapter "Index" updated.

Software	Revision	Date	By	Description
1.00	2	071017	SK	Chapter "Introduction": * Section "Features" enhanced. * Section "Basic concepts" added. * Section "Task and interrupt usage" added. * Section "Further readings" added. Chapter "Running embOS/IP" enhanced. Chapter "API functions": * IP_Init() added. * IP_Task() added. * IP_RxTask() added. * IP_GetVersion() added. * IP_SetLogFilter() added. * IP_SetWarnFilter() added. * IP_Panic() removed. * Structure sockaddr added. * Structure sockaddr_in added. * Structure in_addr added. Chapter "Device driver". * General information updated. * Section "Writing your own driver" added. Chapter "Debugging" added. Chapter "Performance and resource usage" added. Chapter "OS integration" updated.
1.00	1	071002	SK	Product name changed to "embOS/IP": Chapter "API functions": * IP_X_Prepare() renamed to IP_X_Config(). * IP_AddBuffers() added. * IP_ConfTCPSpace() added.
1.00	0	070927	SK	Initial version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler)
- The C programming language
- The target processor
- DOS command line

If you feel that your knowledge of C is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in programm examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table 1.1:

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI
emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS
Real Time Operating System
embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP
TCP/IP stack
embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack
USB device/host stack
A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for micro controllers.

J-Link
JTAG emulator for ARM cores
USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace
USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software
Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Introduction to embOS/IP

This chapter provides an introduction to using embOS/IP. It explains the basic concepts behind embOS/IP.

1.1 What is embOS/IP

embOS/IP is a CPU-independent TCP/IP stack.

embOS/IP is a high-performance library that has been optimized for speed, versatility and small memory footprint.

1.2 Features

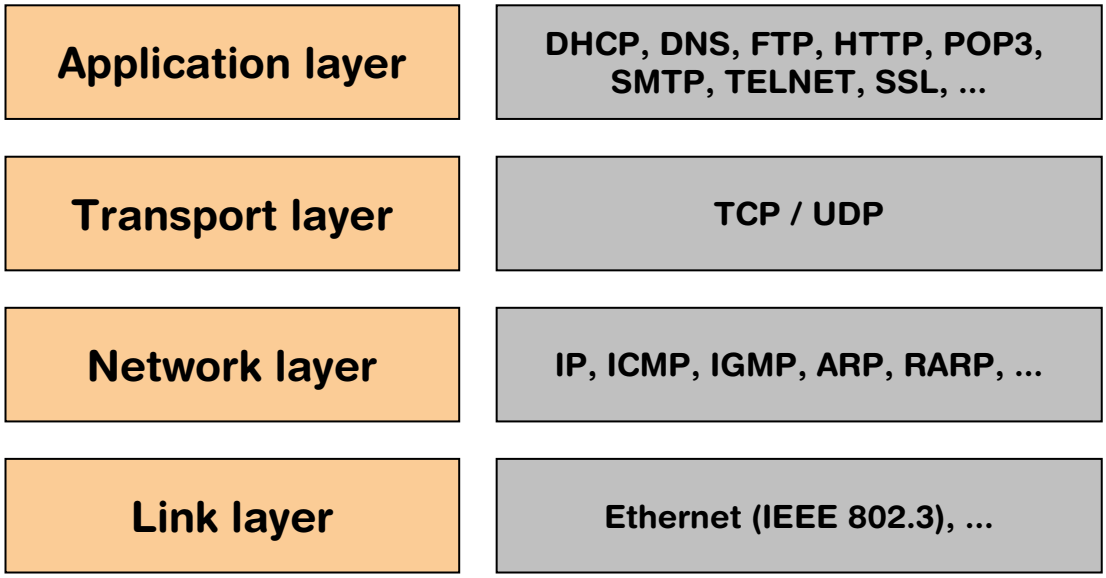
embOS/IP is written in ANSI C and can be used on virtually any CPU.
Some features of embOS/IP:

- Standard socket interface.
- High performance.
- Small footprint.
- No configuration required.
- Runs “out-of-the-box”.
- Very simple network interface driver structure.
- Works seamlessly with embOS in multitasking environment.
- Zero data copy for ultra fast performance.
- Non-blocking versions of all functions.
- Connections limited only by memory availability.
- Delayed ACKs.
- Handling gratuitous ARP packets
- Support for VLAN
- BSD style “keep-alive” option.
- Support for messages and warnings in debug build.
- Drivers for most common Ethernet controllers available.
- Support for driver side (hardware) checksum computation.
- Royalty-free.

1.3 Basic concepts

1.3.1 embOS/IP structure

embOS/IP is organized in different layers, as shown in the following illustration.



A short description of each layer's functionality follows below.

Application layer

The application layer is the interface between embOS/IP and the user application. It uses the embOS/IP API to transmit data over an TCP/IP network. The embOS/IP API provides functions in BSD (Berkeley Software Distribution) socket style, such as `connect()`, `bind()`, `listen()`, etc.

Transport layer

The transport layer provides end-to-end communication services for applications. The two relevant protocols of the Transport layer protocol are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is a reliable connection-oriented transport service. It provides end-to-end reliability, resequencing, and flow control. UDP is a connectionless transport service.

Internet layer

All protocols of the transport layer use the Internet Protocol (IP) to carry data from source host to destination host. IP is a connectionless service, providing no end-to-end delivery guarantees. IP datagrams may arrive at the destination host damaged, duplicated, out of order, or not at all. The transport layer is responsible for reliable delivery of the datagrams when it is required. The IP protocol includes provision for addressing, type-of-service specification, fragmentation and reassembly, and security information.

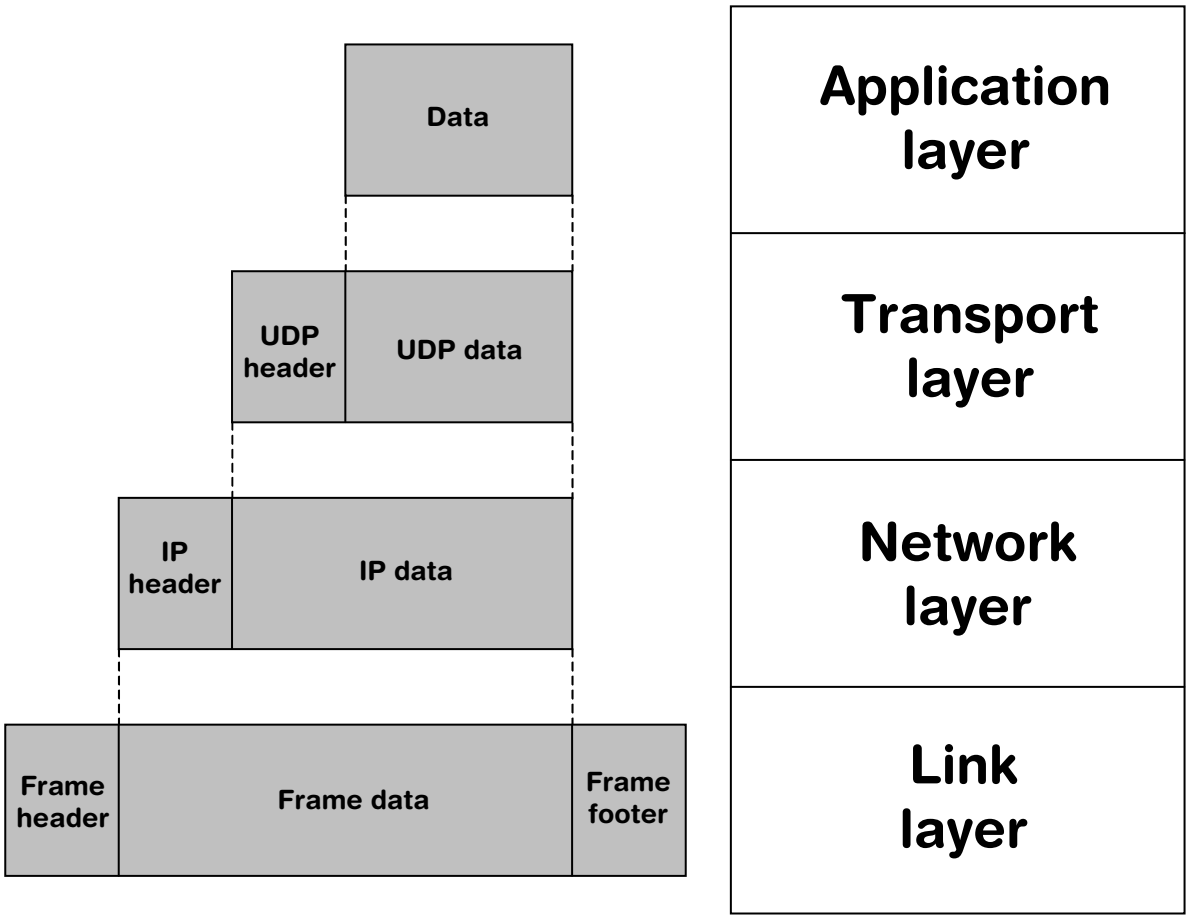
Link layer

The link layer provides the implementation of the communication protocol used to interface to the directly-connected network. A variety of communication protocols have been developed and standardized. The most commonly used protocol is Ethernet (IEEE 802.3). In this version of embOS/IP only Ethernet is supported.

1.3.2 Encapsulation

The four layers structure is defined in *[RFC 1122]*. The data flow starts at the application layer and goes over the transport layer, the network layer, and the link layer. Every protocol adds an protocol-specific header and encapsulates the data and header from the layer above as data. On the receiving side, the data will be extracted in the complementary direction. The opposed protocols do not know which protocol on the above and below layers are used.

The following illustration shows the encapsulation of data within an UDP datagram within an IP packet.



1.4 Tasks and interrupt usage

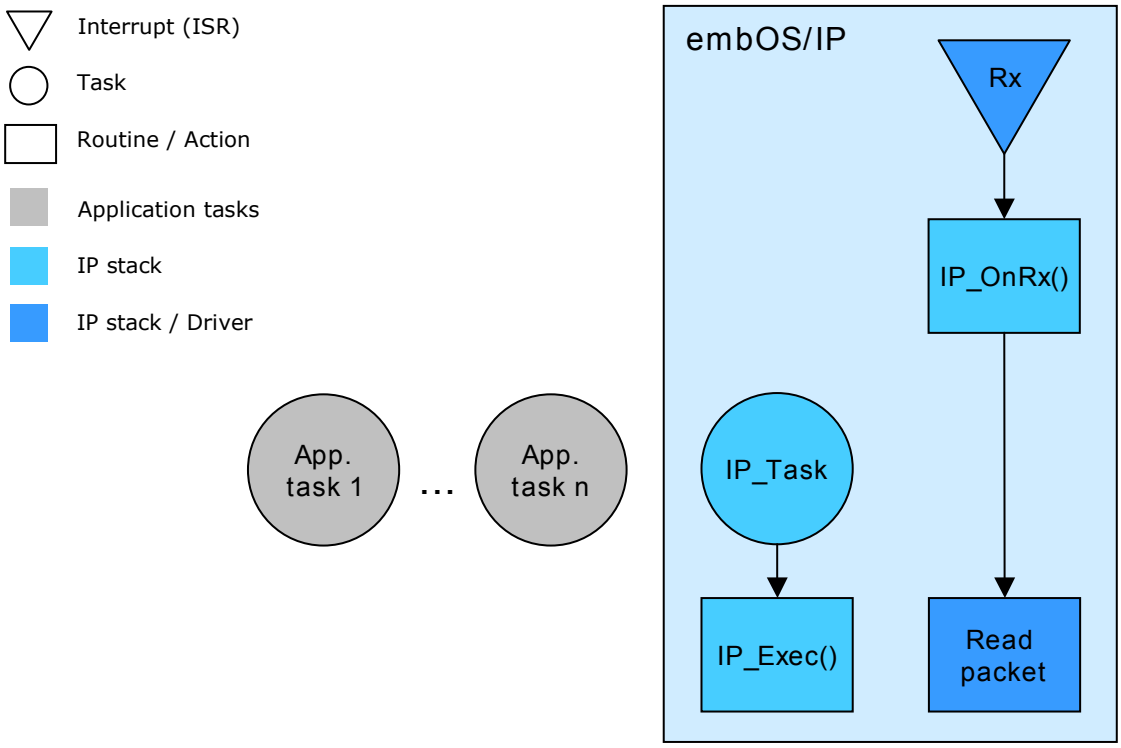
embOS/IP can be used in an application in three different ways.

- One task dedicated to the stack (`IP_Task`)
- Two tasks dedicated to the stack (`IP_Task`, `IP_RxTask`)
- Zero tasks dedicated to the stack (`Superloop`)

The default task structure is one task dedicated to the stack. The priority of the management tasks `IP_Task` (and `IP_RxTask` if available) should be higher then the priority of an application task which uses the stack.

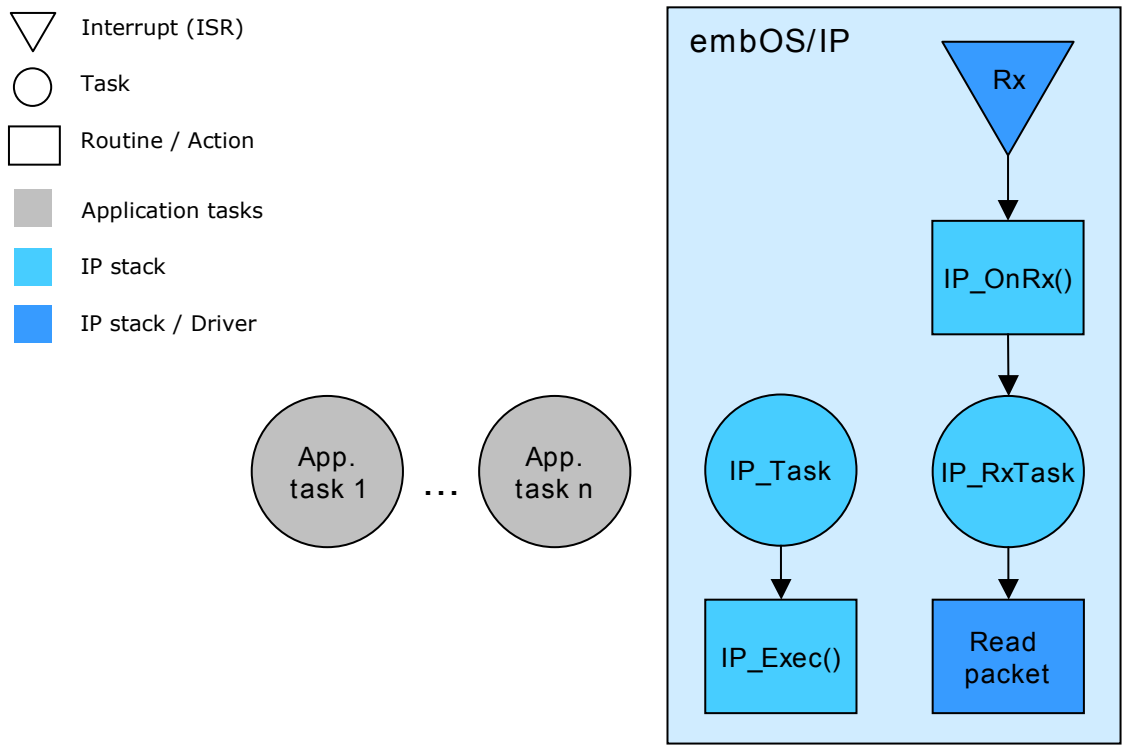
One task dedicated to the stack

To use one task dedicated to the stack is the simplest way to use the TCP/IP stack. It is called `IP_Task` and handles housekeeping operations, resending and handling of incoming packets. The "Read packet" operation is performed from within the ISR. Because the "Read packet" operation is called directly from the ISR, no additional task is required. The length of the interrupt latency will be extended for the time period which is required to process the "Read packet" operation. Refer to *IP_Task()* on page 80 for more information and an example about how to include the `IP_Task` into your embOS project.



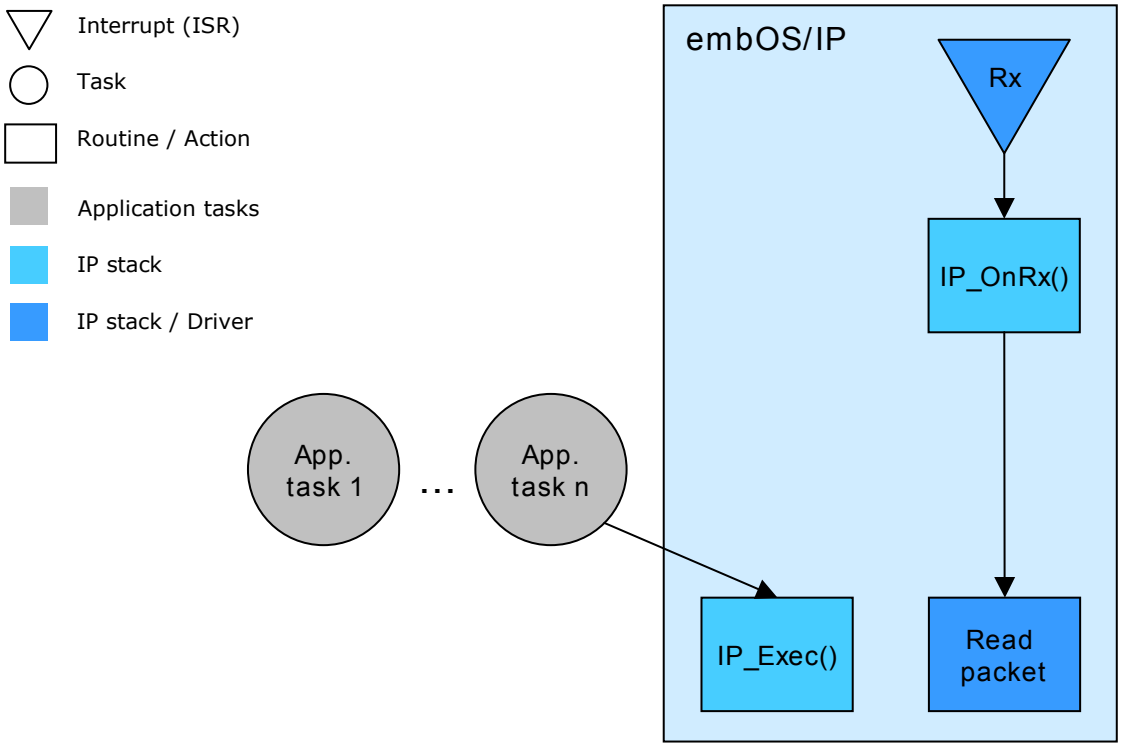
Two tasks dedicated to the stack

Two tasks are dedicated to the stack. The first task is called the `IP_Task` and handles housekeeping operations, resends, and handling of incoming packets. The second is called `IP_RxTask` and handles the "Read packet" operation. `IP_RxTask` is waked up from the interrupt service routine, if new packets are available. The interrupt latency is not extended, because the "Read packet" operation has been moved from the interrupt service routine to `IP_RxTask`. Refer to *IP_Task()* on page 80 and *IP_RxTask()* on page 81 for more information.



Zero tasks dedicated to the stack (Superloop)

embOS/IP can also be used without any additional task for the stack, if an application task calls `IP_Exec()` periodically. The "Read packet" operation is performed from within the ISR. Because the "Read packet" operation is called directly from the ISR, no additional task is required. The length of the interrupt latency will be extended for the time period which is required to process the "Read packet" operation.



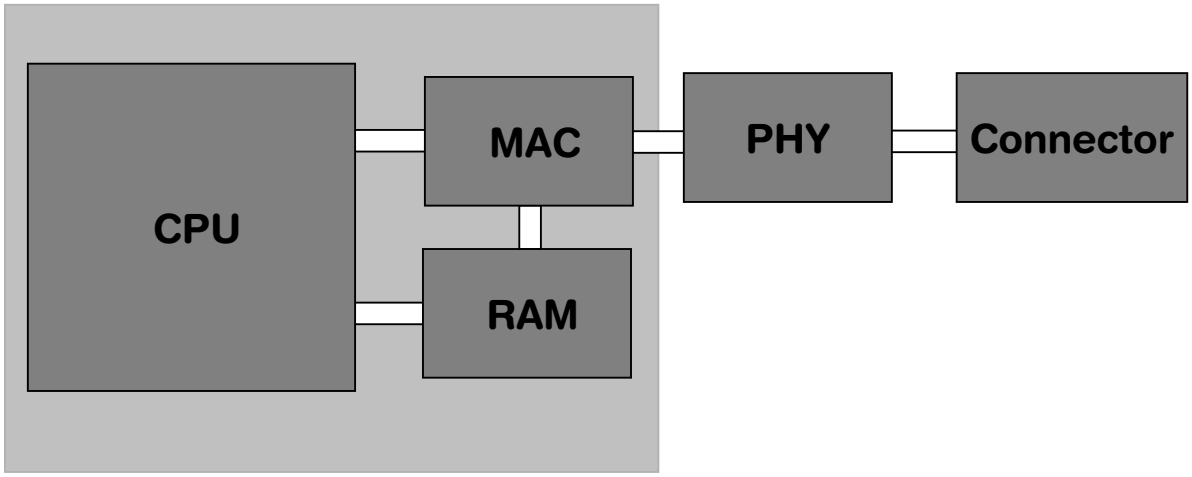
1.5 Background information

1.5.1 Components of an Ethernet system

Main parts of an Ethernet system are the Media Access Controller (MAC) and the Physical device (PHY). The MAC handles generating and parsing physical frames and the PHY handles how this data is actually moved to or from the wire.

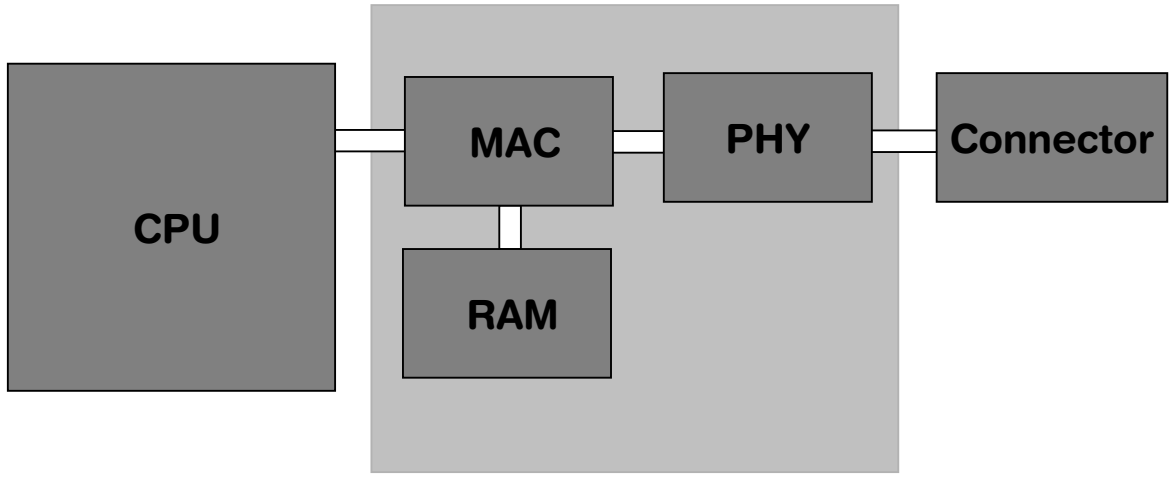
MCUs with integrated MAC

Some modern MCUs (for example, the ATMEL SAM7X or the ST STR912) include the MAC and use the internal RAM to store the Ethernet data. The following block diagram illustrates such a configuration.



External Ethernet controllers with MAC and PHY

Chips without integrated MAC can use fully integrated single chip Ethernet MAC controller with integrated PHY and a general processor interface. The following schematic illustrates such a configuration.



1.5.1.1 MII / RMII: Interface between MAC and PHY

The MAC communicates with the PHY via the Media Independent Interface (MII) or the Reduced Media Independent Interface (RMII). The MII is defined in IEEE 802.3u. The RMII is a subset of the MII and is defined in the RMI specification. The MII/RMII can handle control over the PHY which allows for selection of such transmission criteria as line speed, duplex mode, etc.

In theory, up to 32 PHYs can be connected to a single MAC. In praxis, this is never done; only one PHY is connected. In order to allow multiple PHYs to be connected to a single MAC, individual 5-bit addresses have to be assigned to the different PHYs. If only one PHY is connected, the embOS/IP driver automatically finds the address of it.

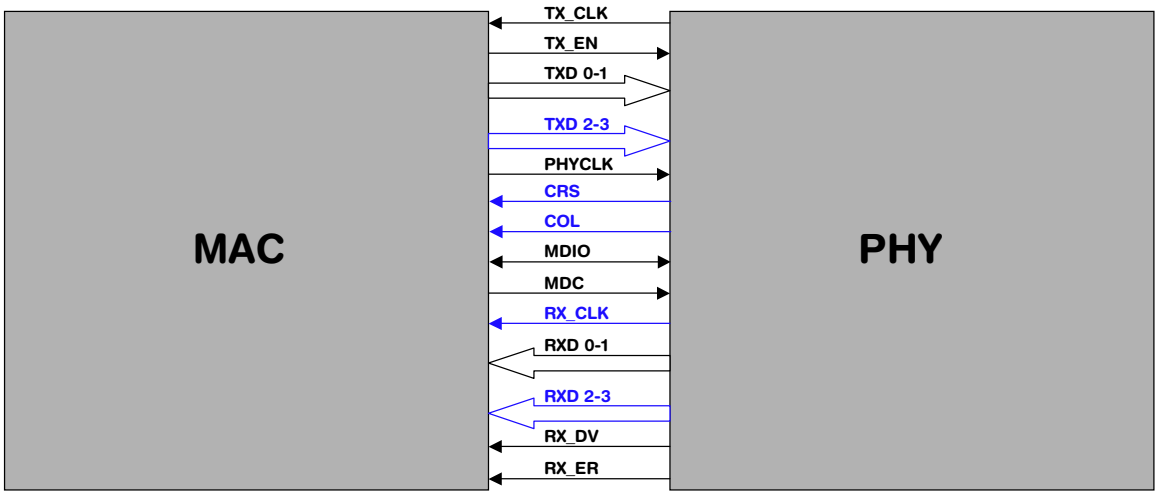
The standard defines 32 16-bit PHY registers. The first 6 are defined by the standard.

Register	Description
BMCR	Basic Mode Control Register
BSR	Basic Mode Status Register
PHYSID1	PHYS ID 1
PHYSID2	PHYS ID 2
ANAR	Auto-Negotiation Advertisement Register
LPAR	Link Partner Ability register

Table 1.1: Standardized registers of the MAC/PHY interface

The drivers automatically recognize any PHY connected, no manual configuration of PHY address is required.

The MII and RMII interface are capable of both 10Mb/s and 100Mb/s data rates as described in the IEEE 802.3u standard.



The intent of the RMII is to provide a reduced pin count alternative to the IEEE 802.3u MII. It uses 2 bits for transmit (TXD0 and TXD1) and two bits for receive (RXD0 and RXD1). There is a Transmit Enable (TX_EN), a Receive Error (RX_ER), a Carrier Sense (CRS), and a 50 MHz Reference Clock (TX_CLK) for 100Mb/s data rate. The pins used by the MII and RMII interfaces are described in the following table.

Signal	MII	RMII
TX_CLK	Transmit Clock (25 MHz)	Reference Clock (50 MHz)
TX_EN	Transmit Enable	Transmit Enable
TXD[0:1]	4-bit Transmit Data	2-bit Transmit Data
TXD[2:3]		N/A
PHYCLK	PHY Clock Output	PHY Clock Output

Table 1.2: MII / RMII comparison

Signal	MII	RMII
CRS	Carrier Sense	N/A
COL	Collision Detect	N/A
MDIO	Management data I/O	Management data I/O
MDC	Data Transfer Timing Reference Clock	Data Transfer Timing Reference Clock
RX_CLK	Receive Clock	N/A
RXD[0:1]	4-bit Receive Data	2-bit Receive Data
RXD[2:3]		N/A
RX_DV	Data Valid	Carrier Sense/Data Valid
RX_ER	Receive Error	Receive Error

Table 1.2: MII / RMII comparison

1.6 Further reading

This guide explains the usage of the embOS/IP protocol stack. It describes all functions which are required to build a network application. For a deeper understanding about how the protocols of the internet protocol suite works use the following references.

The following Request for Comments (RFC) define the relevant protocols of the internet protocol suite and have been used to build the protocol stack. They contain all required technical specifications. The listed books are simpler to read as the RFCs and give a general survey about the interconnection of the different protocols.

1.6.1 Request for Comments (RFC)

RFC#	Description
[RFC 768]	UDP - User Datagram Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc768.txt*
[RFC 791]	IP - Internet Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc791.txt*
[RFC 792]	ICMP - Internet Control Message Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc792.txt*
[RFC 793]	TCP - Transmission Control Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc793.txt*
[RFC 821]	SMTP - Simple Mail Transfer Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc826.txt*
[RFC 826]	ARP - Ethernet Address Resolution Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc826.txt*
[RFC 951]	BOOTP - Bootstrap Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc951.txt*
[RFC 959]	FTP - File Transfer Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc959.txt*
[RFC 1034]	DNS - Domain names - concepts and facilities Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1034.tx*t
[RFC 1035]	DNS - Domain names - implementation and specification Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1035.txt*
[RFC 1042]	IE-EEE - Transmission of IP datagrams over IEEE 802 networks Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1042.txt*
[RFC 1122]	Requirements for Internet Hosts - Communication Layers Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1122.txt*
[RFC 1123]	Requirements for Internet Hosts - Application and Support Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1123.txt*
[RFC 1661]	PPP - Point-to-Point Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1661.txt*
[RFC 1939]	POP3 - Post Office Protocol - Version 3 Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1939.txt*
[RFC 2131]	DHCP - Dynamic Host Configuration Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2131.txt*
[RFC 2616]	HTTP - Hypertext Transfer Protocol -- HTTP/1.1 Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt*

1.6.2 Related books

- [Comer] - Computer Networks and Internets, Douglas E Comer and Ralph E. Droms - ISBN: 978-0131433519
- [Tannenbaum] - Computer Networks, Andrew S. Tannenbaum ISBN: 978-0130661029
- [StevensV1] - TCP/IP Illustrated, Volume 1, W. Richard Stevens ISBN: 978-0201633467.
- [StevensV2] - TCP/IP Illustrated, Volume 2, W. Richard Stevens and Gary R. Wright - ISBN: 978-0201633542.
- [StevensV3] - TCP/IP Illustrated, Volume 3, W. Richard Stevens ISBN: 978-0201634952.

1.7 Development environment (compiler)

The CPU used is of no importance; only an ANSI-compliant C compiler complying with at least one of the following international standard is required:

- ISO/IEC/ANSI 9899:1990 (C90) with support for C++ style comments (//)
- ISO/IEC 9899:1999 (C99)
- ISO/IEC 14882:1998 (C++)

If your compiler has some limitations, let us know and we will inform you if these will be a problem when compiling the software. Any compiler for 16/32/64-bit CPUs or DSPs that we know of can be used; most 8-bit compilers can be used as well.

A C++ compiler is not required, but can be used. The application program can therefore also be programmed in C++ if desired.

Chapter 2

Running embOS/IP on target hardware

This chapter explains how to integrate and run embOS/IP on your target hardware. It explains this process step-by-step.

Integrating embOS/IP

The embOS/IP default configuration is preconfigured with valid values, which matches the requirements of the most applications. embOS/IP is designed to be used with embOS, SEGGER's real-time operating system. We recommend to start with an embOS sample project and include embOS/IP into this project.

We assume that you are familiar with the tools you have selected for your project (compiler, project manager, linker, etc.). You should therefore be able to add files, add directories to the include search path, and so on. In this document the IAR Embedded Workbench® IDE is used for all examples and screenshots, but every other ANSI C toolchain can also be used. It is also possible to use make files; in this case, when we say "add to the project", this translates into "add to the make file".

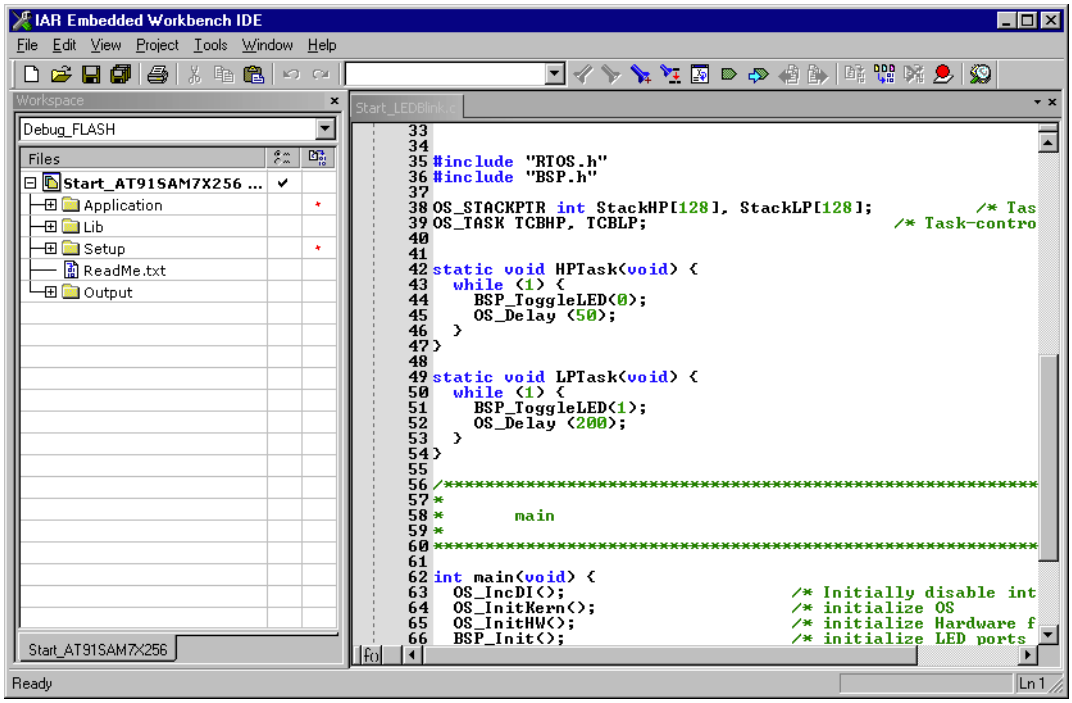
Procedure to follow

Integration of embOS/IP is a relatively simple process, which consists of the following steps:

- Step 1: Open an embOS project and compile it.
- Step 2: Add embOS/IP to the start project
- Step 3: Compile the project

2.1 Step 1: Open an embOS start project

We recommend that you use one of the supplied embOS start projects for your target system. Compile the project and run it on your target hardware.



2.2 Step 2: Adding embOS/IP to the start project

Add all source files in the following directory to your project:

- `Config`
- `IP`
- `UTIL (optional)`

The `Config` folder includes all configuration files of embOS/IP. The configuration files are preconfigured with valid values, which match the requirements of most applications. Add the hardware configuration `IP_Config_<TargetName>.c` supplied with the driver shipment.

If your hardware is currently not supported, use the example configuration file and the driver template to write your own driver. The example configuration file and the driver template is located in the `Sample\Driver\Template` folder.

The `Util` folder is an optional component of the embOS/IP shipment. It contains optimized MCU and/or compiler specific files, for example a special memcopy function.

Replace BSP.c and BSP.h of your embOS start project

Replace the BSP.c source file and the BSP.h header file used in your embOS start project with the one which is supplied with the embOS/IP shipment. Some drivers require a special functions which initializes the network interface of the driver. This function is called `BSP_ETH_Init()`. It is used to enable the ports which are connected to the network hardware. All network interface driver packages include the `BSP.c` and `BSP.h` files irrespective if the `BSP_ETH_Init()` function is implemented.

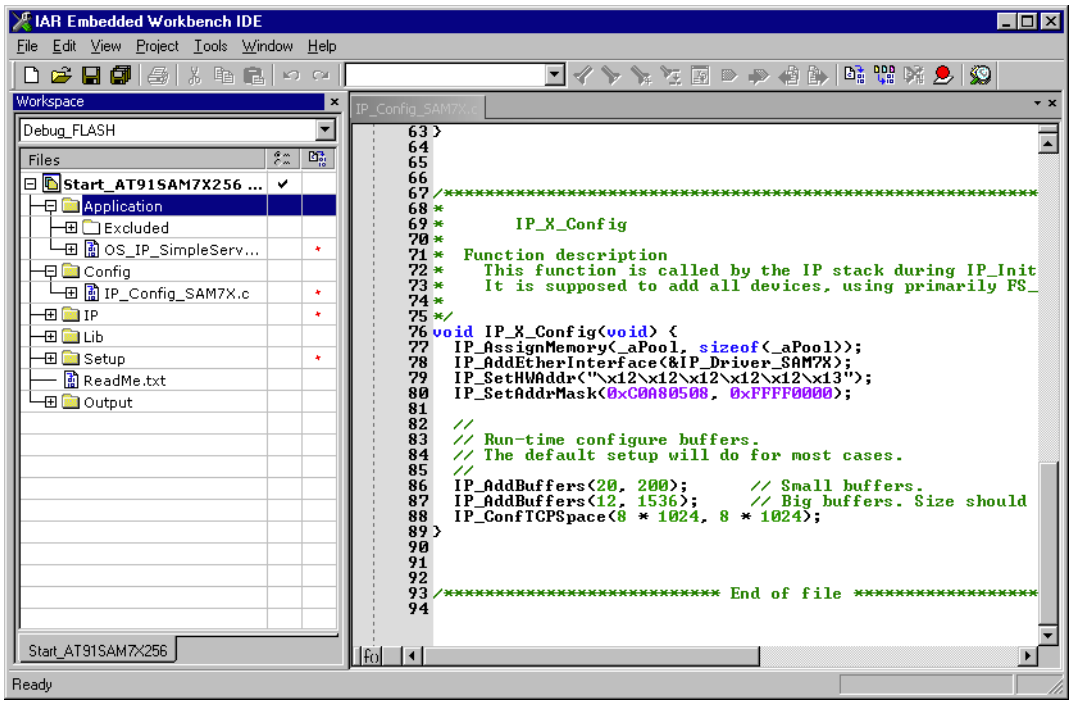
Configuring the include path

The include path is the path in which the compiler looks for include files. In cases where the included files (typically header files, `.h`) do not reside in the same directory as the C file to compile, an include path needs to be set. In order to build the project with all added files, you will need to add the following directories to your include path:

- `Config`
- `Inc`
- `IP`

Select the start application

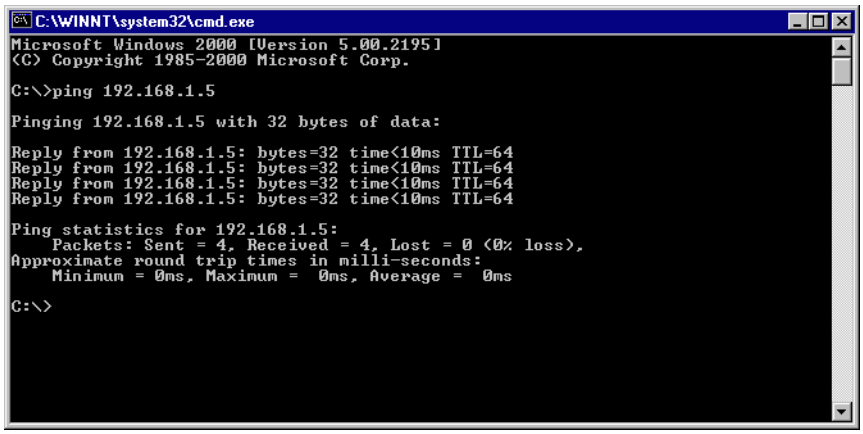
For quick and easy testing of your embOS/IP integration, start with the code found in the folder `Application`. Add one of the applications to your project (for example `OS_IP_SimpleServer.c`).



2.3 Step 3: Build the project and test it

Build the project. It should compile without errors and warnings. If you encounter any problem during the build process, check your include path and your project configuration settings. To test the project, download the output into your target and start the application.

By default, ICMP is activated. This means that you could ping your target. Open the command line interface of your operating system and enter `ping <TargetAddress>`, to check if the stack runs on your target. The target should answer all pings without any error.



Chapter 3

Example applications

In this chapter, you will find a description of each embOS/IP example application.

3.1 Overview

Various example applications for embOS/IP are supplied. These can be used for testing the correct installation and proper function of the device running embOS/IP.

The following start application files are provided:

File	Description
OS_IP_DNSClient.c	Demonstrates the use of the integrated DNS client.
OS_IP_NonBlockingConnect.c	Demonstrates how to connect to a server using non-blocking sockets.
OS_IP_Ping.c	Demonstrates how to send ICMP echo requests and how to process ICMP replies in application.
OS_IP_Shell.c	Demonstrates using the IP-shell to diagnose the IP stack.
OS_IP_SimpleServer.c	Demonstrates setup of a simple server which simply sends back the target system tick for every character received.
OS_IP_SpeedClient_TCP.c	Demonstrates the TCP send and receive performance of the device running embOS/IP. Refer to *embOS/IP speed client (OS_IP_SpeedClient_TCP.c)* on page 38 for detailed information.
OS_IP_Start.c	Demonstrates use of the IP stack without any server or client program. To ping the target, use the command line: `ping <target-ip>` where `<target-ip>` represents the IP address of the target, which depends on the configuration and is usually `192.168.5.1` if the DHCP client is not enabled.
OS_IP_UDPDiscover.c	Demonstrates setup of a simple UDP application which replies to UDP broadcasts. The application sends an answer for every received discover packet. The related host application sends discover packets as UDP broadcasts and waits for the feedback of the targets which are available in the subnet.
OS_IP_UDPDiscoverZeroCopy.c	Demonstrates setup of a simple UDP application which replies to UDP broadcasts. The application uses the the embOS/IP zero-copy interface. It sends an answer for every received discover packet. The related host application sends discover packets as UDP broadcasts and waits for the feedback of the targets which are available in the subnet.

Table 3.1: embOS/IP example applications

The example applications for the target-side are supplied in source code in the `Application` directory.

3.1.1 embOS/IP DNS client (OS_IP_DNSClient.c)

The embOS/IP DNS client resolves a hostname (for example, *segger.com*) to an IP address and outputs the resolved address via terminal I/O.

3.1.2 embOS/IP non-blocking connect (OS_IP_NonBlockingConnect.c)

The embOS/IP non-blocking connect sample demonstrates how to connect to a server using non-blocking sockets. The target tries to connect to TCP server with an non-blocking socket. The sample can be used with any TCP server independent of the application which is listening on the port. The client only opens a TCP connection to the server and closes it without any further communication. The terminal I/O output in your debugger should be similar to the following out:

```
Connecting using non-blocking socket...
Successfully connected after 2ms!
1 of 1 tries were successful.

Connecting using non-blocking socket...
Successfully connected after 1ms!
2 of 2 tries were successful.
```

3.1.3 embOS/IP ping (OS_IP_Ping.c)

The embOS/IP ping sample demonstrates how to send ICMP echo requests and how to process received ICMP packets in your application. A callback function is implemented which outputs a message if an ICMP echo reply or an ICMP echo request has been received.

To test the embOS/IP ICMP implementation, you have to perform the following steps:

1. Customize the `Local defines, configurable` section of `OS_IP_Ping.c`. Change the macro `HOST_TO_PING` accordant to your configuration. For example, if the Windows host PC which you want to ping use the IP address 192.168.5.15, change the `HOST_TO_PING` macro to `0xC0A8050F`.
2. Open the command line interface and enter:
   `ping [IP_ADDRESS _OF_YOUR_TARGET_RUNNING_EMBOSIP]`

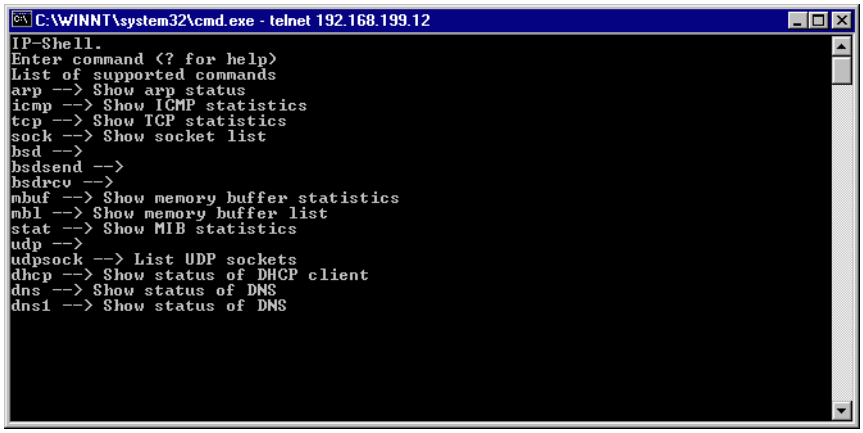
The terminal I/O output in your debugger should be similar to the following out:

```
ICMP echo reply received!
ICMP echo request received!
ICMP echo reply received!
ICMP echo reply received!
ICMP echo reply received!
ICMP echo reply received!
ICMP echo request received!
ICMP echo reply received!
ICMP echo reply received!
ICMP echo reply received!
```

3.1.4 embOS/IP shell (OS_IP_Shell.c)

The embOS/IP shell server is a task which opens TCP-port 23 (telnet) and waits for a connection. The actual shell server is part of the stack, which keep the application program nice and small. The shell server task can be added to any application and should be used to retrieve status information while the target is running. To connect

to the target, use the command line: `telnet <target-ip>` where `<target-ip>` represents the IP address of the target, which depends on the configuration and is usually `192.168.5.230` if the DHCP client is not enabled.



3.1.5 embOS/IP simple server (OS_IP_SimpleServer.c)

Demonstrates setup of a simple server which simply sends back the target system tick for every character received. It opens TCP-port 23 (telnet) and waits for a connection. To connect to the target, use the command line: `telnet <target-ip>` where `<target-ip>` represents the IP address of the target, which depends on the configuration and is usually `192.168.5.230` if the DHCP client is not enabled.

3.1.6 embOS/IP speed client (OS_IP_SpeedClient_TCP.c)

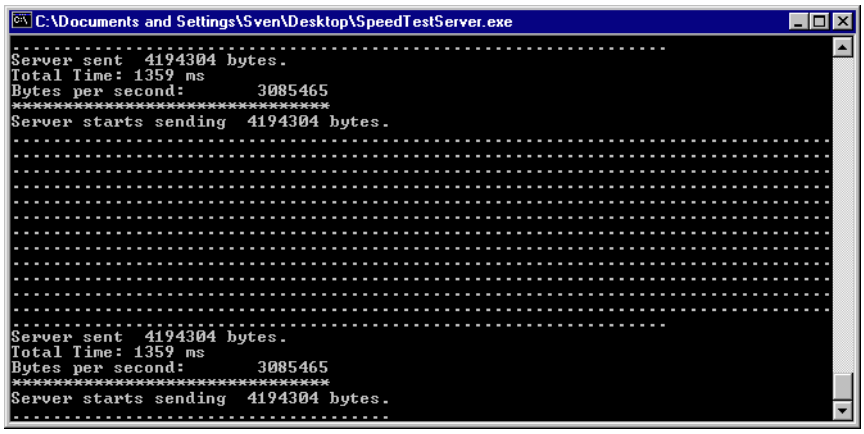
The embOS/IP speed client is a small application to detect the TCP send and receive performance of embOS/IP on your hardware.

3.1.6.1 Running the embOS/IP speed client

To test the embOS/IP performance, you have to perform the following steps:

1. Start the Windows speed test server. The example application for the host-side is supplied as executable and in source code in the `Windows\SpeedTestServer\` directory. To run the speed test server, simply start the executable, for example by double-clicking it or open the supplied Visual C project and compile and start the application.
2. Add `OS_IP_SpeedClient.c` to your project.
3. Customize the `Local defines, configurable` section of `OS_IP_SpeedClient.c`. Change the macro `SERVER_IP_ADDR` accordant to your configuration. For example, if the Windows host PC running the speed test server uses the IP address 192.168.5.15, change the `SERVER_IP_ADDR` macro to `0xC0A8050F`. If you have changed the port which the Windows host application uses to listen, change the macro `SERVER_PORT` accordingly.
4. Build and download the speed client into your target. The target connects to the

server and starts the transmission.



3.1.7 embOS/IP start (OS_IP_Start.c)

Demonstrates use of the IP stack without any server or client program. To ping the target, use the command line: `ping <target-ip>` where `<target-ip>` represents the IP address of the target, which depends on the configuration and is usually `192.168.5.230` if the DHCP client is not enabled.

3.1.8 embOS/IP UDP discover (OS_IP_UDPDiscover.c / OS_IP_UDPDiscoverZeroCopy.c)

To test the embOS/IP UDP discover example, you have to perform the following steps:

1. Start the Windows UDP discover example application. The example application for the host-side is supplied as executable and in source code in the `Windows\UDPDiscover\` directory. To run the UDP discover example, simply start the executable, for example by double-clicking it or open the supplied Visual C project and compile and start the application.
2. Add `OS_IP_UDPDiscover.c` to your project.
3. Customize the `Local defines, configurable` section of `OS_IP_UDPDiscover.c`. By default, the example uses port 50020. If you have changed the port that the Windows host application uses, change the macro `PORT` accordingly.
4. Build and download the UDP discover example into your target. The target sends an answer for every received discover packet. The related host application sends discover packets as UDP broadcasts and waits for the feedback of the targets which are available in the subnet.

Chapter 4

Core functions

In this chapter, you will find a description of each embOS/IP core function.

4.1 API functions

The table below lists the available API functions within their respective categories.

Function	Description
Configuration functions	
IP_AddBuffers()	Adds buffers to the TCP/IP stack.
IP_AddEtherInterface()	Adds an Ethernet interface to the stack.
IP_AllowBackpressure()	Activates back pressure.
IP_AssignMemory()	Assigns memory.
IP_ARP_ConfigAgeout()	Configures the ARP cache timeout.
IP_ARP_ConfigAgeoutNoReply()	Configures the ARP cache timeout for request sent without a reply yet.
IP_ARP_ConfigAgeoutSniff()	Configures the ARP cache timeout for entries sniffed from incoming packets.
IP_ARP_ConfigAllowGratuitousARP()	Configures allow/disallow of using information from gratuitous ARP packets.
IP_ARP_ConfigMaxRetries()	Configures max. ARP request resends.
IP_ARP_ConfigNumEntries()	Configures number of ARP cache entries.
IP_ConfTCPSpace()	Configures the send and receive space.
IP_DNS_SetMaxTTL()	Sets the maximum TTL of a DNS entry.
IP_DNS_SetServer()	Sets the DNS server.
IP_ICMP_Add()	Adds ICMP to the stack.
IP_IGMP_Add()	Adds IGMP to the stack.
IP_IGMP_JoinGroup()	Joins an IGMP group.
IP_IGMP_LeaveGroup()	Leaves an IGMP group.
IP_NI_ConfigPoll()	Select polled mode for the network interface.
IP_NI_SetTxBufferSize()	Configures the Tx buffer size used by the network interface driver.
IP_SetAddrMask()	Sets the address mask of the first interface interface.
IP_SetAddrMaskEx()	Sets the address mask of the selected interface.
IP_SetGWAddr()	Sets the gateway address of the selected interface.
IP_SetHWAddr()	Sets the hardware address of the first interface.
IP_SetHWAddrEx()	Sets the hardware address of the selected interface.
IP_SetMTU()	Sets the maximum transmission unit of an interface.
IP_SetSupportedDuplexModes()	Sets the supported duplex modes.
IP_SetTTL()	Sets the TTL of an IP packet.
IP_SOCKET_SetDefaultOptions()	Sets the socket options which should be enabled by default.
IP_SOCKET_SetLimit()	Sets the maximum number of available sockets.
IP_TCP_Add()	Adds TCP to the stack.
IP_TCP_Set2MSLDelay()	Sets the maximum segment lifetime.
IP_TCP_SetConnKeepaliveOpt()	Sets the keepalive options.
IP_TCP_SetRetransDelayRange()	Sets retransmission delay range.
IP_UDP_Add()	Adds UDP to the stack.

Table 4.1: embOS/IP API function overview

Function	Description
Management functions	
IP_DeInit()	Deinitialization function of the stack.
IP_Init()	Initialization function of the stack.
IP_Task()	Main task for starting the stack.
IP_RxTask()	Reads all available packets and sleeps until a new packet is received.
IP_Exec()	Checks if any packet has been received and handles timers.
Network interface configuration and handling functions	
IP_NI_ConfigPHYAddr()	Configures the PHY address.
IP_NI_ConfigPHYMode()	Configures the PHY mode.
IP_NI_ConfigPoll()	Select polled mode for the network interface.
IP_NI_ForceCaps()	Allows forcing of hardware capabilities.
IP_NI_SetTxBufferSize()	Configures the Tx buffer size used by the network interface driver.
Other IP stack functions	
IP_GetAddrMask()	Returns the IP address and the subnet mask of the device.
IP_GetCurrentLinkSpeed()	Returns the current link speed.
IP_GetCurrentLinkSpeedEx()	Returns the current link speed of the selected interface.
IP_GetGWAddr()	Returns the gateway address of the device.
IP_GetHWAddr()	Returns the hardware address (MAC) of the device.
IP_GetIPAddr()	Returns the IP address of the device.
IP_GetIPPacketInfo()	Returns the start address of the data part of an IP packet.
IP_GetRawPacketInfo()	Returns the start address of the raw data part of an IP packet.
IP_GetVersion()	Returns the version number of embOS/IP.
IP_ICMP_SetRxHook()	Sets a hook function which will be called if target receives a ping packet.
IP_IFaceIsReady()	Checks if the interface is ready.
IP_IFaceIsReadyEx()	Checks if the specified interface is ready.
IP_PrintIPAddr()	Convert an 4 byte IP address to a dots-and-number string.
IP_SendPacket()	Sends a user defined packet on the interface.
IP_SendPing()	Sends an ICMP Echo Request.
IP_SendPingEx()	Sends an ICMP Echo Request.
IP_SetRxHook()	Sets a hook function that handles all received packets.

Table 4.1: embOS/IP API function overview (Continued)

4.2 Configuration functions

4.2.1 IP_AddBuffers()

Description

Adds buffers to the TCP/IP stack. This is a configuration function, typically called from `IP_X_Config()`. It needs to be called 2 times, one per buffer size.

Prototype

```
void IP_AddBuffers ( int NumBuffers,
                     int BytesPerBuffer );
```

Parameter

Parameter	Description
NumBuffers	[IN] The number of buffers.
BytesPerBuffer	[IN] Size of buffers in bytes.

Table 4.2: IP_AddBuffers() parameter list

Additional information

embOS/IP requires small and large buffers. We recommend to define the size of the big buffers to 1536 to allow a full Ethernet packet to fit. The small buffers are used to store packets which encapsulates no or few application data like protocol management packets (TCP SYNs, TCP ACKs, etc.). We recommend to define the size of the small buffers to 256 bytes.

Example

```
IP_AddBuffers(20, 256);          // 20 small buffers, each 256 bytes.
IP_AddBuffers(12, 1536);         // 12 big buffers, each 1536 bytes.
```


4.2.2 IP_AddEtherInterface()

Description

Adds an Ethernet interface.

Prototype

```
void IP_AddEtherInterface ( const IP_HW_DRIVER * pDriver );
```

Parameter

Parameter	Description
pDriver	[IN] A pointer to a network interface driver structure.

Table 4.3: IP_AddEtherInterface() parameter list

Additional information

Refer to *Available network interface drivers* on page 219 for a list of available network interface drivers.

Example

```
IP_AddEtherInterface(&IP_Driver_SAM7X);     // Add Ethernet driver for your hardware
```


4.2.3 IP_AllowBackpressure()

Description

Allows back pressure if the driver supports this feature.

Prototype

```
void IP_AllowBackpressure ( int v );
```

Parameter

Parameter	Description
v	[IN] Zero to disable, 1 to enable back pressure.

Table 4.4: IP_AllowBackPressure() parameter list

Additional information

Back pressure is a window-based flow control mechanism for the half-duplex mode. It is a sort of feedback-based congestion control mechanism. The intent of this mechanism is to prevent loss by providing back pressure to the sending NIC on ports that are going too fast to avoid loss. Back pressure is enabled by default.

4.2.4 IP_AssignMemory()

Description

Assigns memory to the TCP/IP stack.

Prototype

```
void IP_AssignMemory ( U32 * pMem,
                       U32   NumBytes );
```

Parameter

Parameter	Description
pMem	[IN] A pointer to the start of the memory region which should be assigned.
NumBytes	[IN] Number of bytes which should be assigned.

Table 4.5: IP_AssignMemory() parameter list

Additional information

`IP_AssignMemory()` should be the first function which is called in `IP_X_Config()`. The amount of RAM required depends on the configuration and the respective application purpose. The assigned memory pool is required for the socket buffers, memory buffers, etc.

Example

```
#define ALLOC_SIZE      0x8000     // Size of memory dedicated to the stack in bytes
U32 _aPool[ALLOC_SIZE / 4];        // Memory area used by the stack.

IP_AssignMemory(_aPool, sizeof(_aPool));
```


4.2.5 IP_ARP_ConfigAgeout()

Description

Configures the timeout for cached ARP entries.

Prototype

```
void IP_ARP_ConfigAgeout ( U32 Ageout );
```

Parameter

Parameter	Description
Ageout	[IN] Timeout in ms after which an entry is deleted from the ARP cache. Default: 120s.

Table 4.6: IP_ARP_ConfigAgeout() parameter list

4.2.6 IP_ARP_ConfigAgeoutNoReply()

Description

Configures the timeout for an ARP entry that has been added due to sending an ARP request to the network that has not been answered yet.

Prototype

```
void IP_ARP_ConfigAgeoutNoReply ( U32 Ageout );
```

Parameter

Parameter	Description
Ageout	[IN] Timeout in ms after which an entry is deleted in case we are still waiting for an ARP response. Default: 3s.

Table 4.7: IP_ARP_ConfigAgeoutNoReply() parameter list

4.2.7 IP_ARP_ConfigAgeoutSniff()

Description

Configures the timeout for cached ARP entries that have been cached from incoming packets instead from sending an ARP request.

Prototype

```
void IP_ARP_ConfigAgeoutSniff ( U32 Ageout );
```

Parameter

Parameter	Description
Ageout	[IN] Timeout in ms after which an entry is deleted from the ARP cache.

Table 4.8: IP_ARP_ConfigAgeoutSniff() parameter list

4.2.8 IP_ARP_ConfigAllowGratuitousARP()

Description

Configures if gratuitous ARP packets from other network members are allowed to update the ARP cache.

Prototype

```
void IP_ARP_AllowGratuitousARP ( U8 OnOff );
```

Parameter

Parameter	Description
OnOff	[IN] 0: Off; 1: On. Default: On.

Table 4.9: IP_ConfigAllowGratuitousARP() parameter list

Additional information

Gratuitous ARP packets allow the network to update itself by sending out informations about changes regarding IP and hardware ID assignments. As this behaviour helps the network to become more stable and helps to manage itself it is on by default.

In case you consider gratuitous ARP packets as a security risk `IP_ARP_ConfigAllowGratuitousARP()` can be used to disallow this behaviour.

4.2.9 IP_ARP_ConfigMaxRetries()

Description

Configures how often an ARP request is resent before considering the request failed.

Prototype

```
void IP_ARP_ConfigConfigMaxRetries ( unsigned Retries );
```

Parameter

Parameter	Description
Retries	[IN] Number of retries for sending an ARP request.

Table 4.10: IP_ARP_ConfigMaxRetries() parameter list

4.2.10 IP_ARP_ConfigNumEntries()

Description

Configures the maximum number of possible entries in the ARP cache.

Prototype

```
int IP_ARP_ConfigNumEntries ( unsigned NumEntries );
```

Parameter

Parameter	Description
NumEntries	[IN] Number of max. entries in ARP cache list.

Table 4.11: IP_ARP_ConfigNumEntries() parameter list

Retrurn value

 0: O.K., the stack will try to allocate the requested number of entries.
-1: Error, called after IP_Init().

Additional information

`IP_ARP_ConfigNumEntries()` has to be called before IP_Init().

4.2.11 IP_ConfTCPSpace()

Description

Configures the size of the TCP send and receive window size.

Prototype

```
void IP_ConfTCPSpace ( unsigned SendSpace,
                       unsigned RecvSpace );
```

Parameter

Parameter	Description
SendSpace	[IN] Size of the send window.
RecvSpace	[IN] Size of the receive window.

Table 4.12: IP_ConfTCPSpace() parameter list

Additional information

The receive window size is the amount of unacknowledged data a sender can send to the receiver on a particular TCP connection before it gets an acknowledgment.

4.2.12 IP_DNS_SetMaxTTL()

Description

Sets the maximum Time To Live (TTL) of a DNS entry in seconds.

Prototype

```
void IP_DNS_SetMaxTTL( U32 TTL );
```

Parameter

Parameter	Description
TTL	[IN] Maximum TTL of a DNS entry in seconds.

Table 4.13: IP_DNS_SetMaxTTL() parameter list

Additional information

The real TTL is the minimum of TTL and the TTL specified by the DNS server for the entry. The embOS/IP default for the maximum TTL of an DNS entry is 600 seconds.

4.2.13 IP_DNS_SetServer()

Description

Sets the DNS server that should be used.

Prototype

```
void IP_DNS_SetServer ( U32 DNSServerAddr );
```

Parameter

Parameter	Description
DNSServerAddr	[IN] Address of DNS server.

Table 4.14: IP_DNS_SetServer() parameter list

Additional information

If a DHCP server is used for configuring your target, `IP_DNS_SetServer()` should not be called. The DNS server settings are normally part of the DHCP configuration setup. The DNS server has to be defined before calling `gethostbyname()` to resolve an internet address. Refer to *gethostbyname()* on page 116 for detailed information about resolving an internet address.

4.2.14 IP_ICMP_Add()

Description

Adds ICMP to the stack.

Prototype

```
void IP_ICMP_Add ( void );
```

Additional information

`IP_ICMP_Add()` adds ICMP to the stack. The function should be called during the initialization of the stack. In the supplied sample configuration files `IP_ICMP_Add()` is called from `IP_X_Config()`. If you remove the call of `IP_ICMP_Add()`, the ICMP code will not be available in your application.

4.2.15 IP_IGMP_Add()

Description

Adds IGMP to the stack.

Prototype

```
void IP_IGMP_Add ( void );
```

Additional information

`IP_IGMP_Add()` adds IGMP (Internet Group Management Protocol) to the stack. The function should be either called during the initialization of the stack by adding it to your `IP_X_Config()` or should be called after `IP_Init()`. If you remove the call of `IP_IGMP_Add()`, the ICMP code will not be available in your application.

4.2.16 IP_IGMP_JoinGroup()

Description

Joins an IGMP group.

Prototype

```
void IP_IGMP_JoinGroup ( unsigned IFace,
                         IP_ADDR  GroupIP );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based index of available interfaces.
GroupIP	[IN] IGMP group IP addr.

Table 4.15: IP_IGMP_JoinGroup() parameter list

Additional information

Calling this function should be only done after `IP_init()` as we relay on an already configured HW addr.

Multicast is a technique to distribute a packet to multiple receivers in a network by sending only one packet. Handling of who will receive the packet is not done by the sender but instead is done by network hardware such as routers or switched hubs that will dupplicate the packet and send it to everyone that participates the chosen group.

The target does not actively participate by sending a join request. The network hardware periodically broadcasts membership querys throughout the network that have to be answered with a membership report in case we want to participate in the queried group.

Example

```
/* Excerpt from IP.h */
#define IP_IGMP_MCAST_ALLHOSTS_GROUP  0xE0000001uL  // 224.0.0.1
#define IP_IGMP_MCAST_ALLRPTS_GROUP   0xE0000016uL  // 224.0.0.22, IGMPv3

/* Excerpt from the UPnP code */
#define SSDP_IP  0xEFFFFFFA  // Simple service discovery protocol IP, 239.255.255.250

IP_IGMP_Add();  // IGMP is needed for UPnP
//
// Join IGMP ALLHOSTS group and IGMP group for SSDP
//
IP_IGMP_JoinGroup(0, IP_IGMP_MCAST_ALLHOSTS_GROUP);
IP_IGMP_JoinGroup(0, SSDP_IP);
```


4.2.17 IP_IGMP_LeaveGroup()

Description

Leaves an IGMP group.

Prototype

```
void IP_IGMP_LeaveGroup ( unsigned IFace,
                          IP_ADDR  GroupIP );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based index of available interfaces.
GroupIP	[IN] IGMP group IP addr.

Table 4.16: IP_IGMP_LeaveGroup() parameter list

Additional information

The target does not actively participate by sending a leave request. Instead the target will change its filters to no longer receiving IGMP membership querys and will then be removed from the list of participants of the network hardware after a timeout.

Example

```
/* Excerpt from IP.h */
#define IP_IGMP_MCAST_ALLHOSTS_GROUP  0xE0000001uL  // 224.0.0.1
#define IP_IGMP_MCAST_ALLRPTS_GROUP   0xE0000016uL  // 224.0.0.22, IGMPv3

/* Sample for leaving IGMP groups used for UPnP */
#define SSDP_IP  0xEFFFFFFA  // Simple service discovery protocol IP, 239.255.255.250

//
// Leave IGMP ALLHOSTS group and IGMP group for SSDP
//
IP_IGMP_LeaveGroup(0, IP_IGMP_MCAST_ALLHOSTS_GROUP);
IP_IGMP_LeaveGroup(0, SSDP_IP);
```


4.2.18 IP_SetAddrMask()

Description

Sets the IP address and subnet mask of the first interface of the stack (interface 0).

Prototype

```
void IP_SetAddrMask ( U32 Addr,
                      U32 Mask );
```

Parameter

Parameter	Description
Addr	[IN] 4-byte IPv4 address.
Mask	[IN] Subnet mask.

Table 4.17: IP_SetAddrMask() parameter list

Additional information

The address mask should only be set if no DHCP server is used to obtain IP address, subnet mask and default gateway. Refer to chapter *DHCP client* on page 169 for detailed information about the usage of the embOS/IP DHCP client.

Example

```
IP_SetAddrMask(0xC0A80505, 0xFFFF0000);     // IP: 192.168.5.5
                                            // Subnet mask: 255.255.0.0
```


4.2.19 IP_SetAddrMaskEx()

Description

Sets the IP address and subnet mask of an interface.

Prototype

```
void IP_SetAddrMaskEx  ( U8  IFace,
                         U32 Addr,
                         U32 Mask );
```

Parameter

Parameter	Description
IFace	[IN] Interface Id.
Addr	[IN] 4-byte IPv4 address.
Mask	[IN] Subnet mask.

Table 4.18: IP_SetAddrMaskEx() parameter list

Additional information

The address mask should only be set if no DHCP server is used to obtain IP address, subnet mask and default gateway. Refer to chapter *DHCP client* on page 169 for detailed information about the usage of the embOS/IP DHCP client.

Example

```
IP_SetAddrMaskEx(0, 0xC0A80505, 0xFFFF0000);      // Interface: 0
                                                  // IP: 192.168.5.5
                                                  // Subnet mask: 255.255.0.0
```


4.2.20 IP_SetGWAddr()

Description

Sets the default gateway address of the selected interface.

Prototype

```
void IP_SetGWAddr  ( U8  IFace,
                     U32 Addr );
```

Parameter

Parameter	Description
IFace	[IN] Interface Id.
Addr	[IN] 4-byte gateway address.

Table 4.19: IP_SetGWAddrEx() parameter list

Additional information

The address mask should only be set if no DHCP server is used to obtain IP address, subnet mask and default gateway. Refer to chapter *DHCP client* on page 169 for detailed information about the usage of the embOS/IP DHCP client.

Example

```
IP_SetGWAddr(0, 0xC0A80101);      // Interface: 0
                                  // IPv4 address of the GW: 192.168.1.1
```


4.2.21 IP_SetHWAddr()

Description

Sets the Media Access Control address (MAC) of the first interface (interface 0).

Prototype

```
void IP_SetHWAddr( const U8 * pHWAddr );
```

Parameter

Parameter	Description
pHWAddr	[IN] 6-byte MAC address.

Table 4.20: IP_SetHWAddr() parameter list

Additional information

The MAC address needs to be unique for production units.

Example

```
IP_SetHWAddr("\x00\x22\x33\x44\x55\x66");
```


4.2.22 IP_SetHWAddrEx()

Description

Sets the Media Access Control address (MAC) of the selected interface.

Prototype

```
void IP_SetHWAddr( const U8 * pHWAddr );
```

Parameter

Parameter	Description
pHWAddr	[IN] 6-byte MAC address.

Table 4.21: IP_SetHWAddrEx() parameter list

Additional information

The MAC address needs to be unique for production units.

Example

```
IP_SetHWAddrEx(0, "\x00\x22\x33\x44\x55\x66");
```


4.2.23 IP_SetMTU()

Description

Allows to set the Maximum Transmission Unit (MTU) of the selected interface.

Prototype

```
void IP_SetMTU( U8  IFace,
                U32 Mtu );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based index of available network interfaces.
Mtu	[IN] Size of maximum transmission unit in bytes.

Table 4.22: IP_SetMTU() parameter list

Additional information

The Maximum Transmission Unit is the MTU from an IP standpoint, so the size of the IP-packet without local net header. A typical value for ethernet is 1500, since the maximum size of an Ethernet packet is 1518 bytes. Since Ethernet uses 12 bytes for MAC addresses, 2 bytes for type and 4 bytes for CRC, 1500 bytes "payload" remain. The minimum size of the MTU is 576 according to RFC 879. Refer to *[RFC 879] - TCP - The TCP Maximum Segment Size and Related Topics* for more information about the MTU.

A smaller MTU size is effective for TCP connections only, it does not affect UDP connections. All TCP connections are guaranteed to work with any MTU in the permitted range of 576 - 1500 bytes. The advantage of a smaller MTU is that smaller packets are sent in TCP communication, resulting in reduced RAM requirements, especially if the window size is also reduced. The disadvantage is a loss of communication speed.

Note: In the supplied embOS/IP example configurations, the MTU is used to configure the maximum packet size that the stack can handle. This means that if you lower the MTU (for example, set it to 576 bytes), the stack can only handle packets up to that size. If you plan to use larger UDP packets, change the configuration according to your requirements. For further information about the configuration of the stack, refer to *Configuring embOS/IP* on page 255.

4.2.24 IP_SetSupportedDuplexModes()

Description

Allows to set the allowed Duplex modes.

Prototype

```
int IP_SetSupportedDuplexModes( unsigned Unit,
                                unsigned DuplexMode);
```

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.
DuplexMode	[IN] OR-combination of one or more of the following valid values.

Table 4.23: IP_SetSupportedDuplexModes() parameter list

Valid values for parameter DuplexMode

Value	Description
IP_PHY_MODE_10_HALF	Support 10 Mbit half-duplex
IP_PHY_MODE_10_FULL	Support 10 Mbit full-duplex
IP_PHY_MODE_100_HALF	Support 100 Mbit half-duplex
IP_PHY_MODE_100_FULL	Support 100 Mbit full-duplex

4.2.25 IP_SetTTL()

Description

Sets the default value for the Time-To-Live IP header field.

Prototype

```
void IP_SetTTL ( int v );
```

Parameter

Parameter	Description
v	[IN] Time-To-Live value.

Table 4.24: IP_SetTTL() parameter list

Additional information

By default, the TTL (Time-To-Live) is 64. The TTL field length of the IP is 8 bits. The maximum value of the TTL field is therefore 255.

4.2.26 IP_SOCKET_SetDefaultOptions()

Description

Allows to set the maximum transmission unit (MTU) of an interface.

Prototype

```
void IP_SOCKET_SetDefaultOptions ( U16 v );
```

Parameter

Parameter	Description
v	[IN] Socket options which should be enabled. By default, keepalive (SO_KEEPALIVE) socket option is enabled. Refer to *setsockopt()* on page 132 for a list of supported socket options.

Table 4.25: IP_SOCKET_SetDefaultOptions() parameter list

4.2.27 IP_SOCKET_SetLimit()

Description

Sets the maximum number of available sockets.

Prototype

```
void IP_SOCKET_SetLimit ( unsigned Limit );
```

Parameter

Parameter	Description
Limit	[IN] Sets a limit on number of sockets which can be created. The embOS/IP default is 0 which means that no limit is set.

Table 4.26: IP_SOCKET_SetLimit() parameter list

4.2.28 IP_TCP_Add()

Description

Adds TCP to the stack.

Prototype

```
void IP_TCP_Add ( void );
```

Additional information

`IP_TCP_Add()` adds TCP to the stack. The function should be called during the initialization of the stack. In the supplied sample configuration files `IP_TCP_Add()` is called from `IP_X_Config()`. If you remove the call of `IP_TCP_Add()`, the TCP code will not be available in your application.

4.2.29 IP_TCP_Set2MSLDelay()

Description

Sets the maximum segment lifetime (MSL).

Prototype

```
void IP_TCP_Set2MSLDelay( unsigned v );
```

Parameter

Parameter	Description
v	[IN] Maximum segment lifetime. The embOS/IP default is 2 seconds.

Table 4.27: IP_TCP_Set2MSLDelay() parameter list

Additional information

The maximum segment lifetime is the amount of time any segment can exist in the network before being discarded. This time limit is constricted. When TCP performs an active close the connection must stay in TIME_WAIT (2MSL) state for twice the MSL after sending the final ACK.

Refer to *[RFC 793] - TCP - Transmission Control Protocol* for more information about TCP states.

4.2.30 IP_TCP_SetConnKeepaliveOpt()

Description

Sets the keepalive options.

Prototype

```
void IP_TCP_SetConnKeepaliveOpt( U32 Init,
                                 U32 Idle,
                                 U32 Period,
                                 U32 MaxRep );
```

Parameter

Parameter	Description
Init	[IN] Maximum time for TCP-connection open (response to SYN) in ms. The embOS/IP default is 20 seconds.
Idle	[IN] Time of TCP-inactivity before first keepalive probe is sent in ms. The embOS/IP default is 60 seconds.
Period	[IN] Time of TCP-inactivity between keepalive probes in ms. The embOS/IP default is 10 seconds.
MaxRep	[IN] Number of keepalive probes before we give up and close the connection. The embOS/IP default is 8 repetitions.

Table 4.28: IP_TCP_SetConnKeepaliveOpt() parameter list

Additional information

Keepalives are not part of the TCP specification, since they can cause good connections to be dropped during transient failures. For example, if the keepalive probes are sent during the time that an intermediate router has crashed and is rebooting, TCP will think that the client's host has crashed, which is not what has happened. Nevertheless, the keepalive feature is very useful for embedded server applications that might tie up resources on behalf of a client, and want to know if the client host crashes.

4.2.31 IP_TCP_SetRetransDelayRange()

Description

Sets the retransmission delay range.

Prototype

```
void IP_TCP_SetRetransDelayRange( unsigned RetransDelayMin,
                                  unsigned RetransDelayMax );
```

Parameter

Parameter	Description
RetransDelayMin	[IN] Minimum time before first retransmission. The embOS/IP default is 200 ms.
RetransDelayMax	[IN] Maximum time to wait before a retransmission. The embOS/IP default is 5 seconds.

Table 4.29: IP_TCP_SetRetransDelayRange() parameter list

Additional information

TCP is a reliable transport layer. One of the ways it provides reliability is for each end to acknowledge the data it receives from the communication partner. But data segments and acknowledgments can get lost. TCP handles this by setting a timeout when it sends data, and if the data is not acknowledged when the timeout expires, it retransmits the data. The timeout and retransmission is the measurement of the round-trip time (RTT) experienced on a given connection. The RTT can change over time, as routes might change and as network traffic changes, and TCP should track these changes and modify its timeout accordingly. `IP_TCP_SetRetransDelayRange()` should be called if the default limits are not sufficient for your application.

4.2.32 IP_UDP_Add()

Description

Adds UDP to the stack.

Prototype

```
void IP_UDP_Add ( void );
```

Additional information

`IP_UDP_Add()` adds UDP to the stack. The function should be called during the initialization of the stack. In the supplied sample configuration files `IP_UDP_Add()` is called from `IP_X_Config()`. If you remove the call of `IP_UDP_Add()`, the UDP code will not be available in your application.

4.3 Management functions

4.3.1 IP_DeInit()

Description

De-initializes the TCP/IP stack.

Prototype

```
void IP_DeInit ( void );
```

Additional information

`IP_DeInit()` de-initializes the IP stack. This function should be the very last embOS/IP function called and is typically not needed if you do not need to shutdown your whole application for a special reason.

Example

```
#include "IP.h"

void main(void) {
  IP_Init();
  /*
   *  Use the stack
   */
  IP_DeInit();
}
```


4.3.2 IP_Init()

Description

Initializes the TCP/IP stack.

Prototype

```
void IP_Init ( void );
```

Additional information

`IP_Init()` initializes the IP stack and creates resources required for an OS integration. This function must be called before any other embOS/IP function is called.

Example

```
#include "IP.h"

void main(void) {
  IP_Init();
  /*
   *  Use the stack
   */
}
```


4.3.3 IP_Task()

Description

Main task for starting the stack. After startup, it settles into a loop handling received packets. This loop sleeps until a packet has been queued in the receive queue; then it should be awakened by the driver which queued the packet.

Prototype

```
void IP_Task ( void );
```

Additional information

Implementing this task is the simplest way to include embOS/IP into your project. Typical stack usage is approximately 440 bytes. To be on the safe side set the size of the task stack to 1024 bytes.

Note: The priority of task `IP_Task` should be higher then the priority of an application task which uses the stack.

Example

```
#include <stdio.h>

#include "RTOS.h"
#include "BSP.h"
#include "IP.h"
#include "IP_Int.h"

static OS_STACKPTR int _Stack0[512];     // Task stacks
static OS_TASK         _TCB0;            // Task-control-blocks
static OS_STACKPTR int _IPStack[1024];   // Task stacks
static OS_TASK         _IPTCB;           // Task-control-blocks

/*********************************************************************
*
*       MainTask
*/
void MainTask(void);
void MainTask(void) {
  printf("**************************************\nProgram start\n");
  IP_Init();
  OS_SetPriority(OS_GetTaskID(), 255);    // This task has highest prio!
  OS_CREATETASK(&_IPTCB, "IP_Task", IP_Task, 150, _IPStack);
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}

/*********************************************************
*
*       main
*/
void main(void) {
  BSP_Init();
  BSP_SetLED(0);
  OS_IncDI();           /* Initially disable interrupts  */
  OS_InitKern();        /* initialize OS                 */
  OS_InitHW();          /* initialize Hardware for OS    */
  OS_CREATETASK(&_TCB0, "MainTask", MainTask, 100, _Stack0);
  OS_Start();
}
```


4.3.4 IP_RxTask()

Description

The task reads all available packets from the network interface and sleeps until a new packet is received.

Prototype

```
void IP_RxTask ( void );
```

Additional information

This task is optional. Refer to *Tasks and interrupt usage* on page 19 for detailed information about the task and interrupt handling of embOS/IP. Typical stack usage is approximately 150 bytes. To be on the safe side set the size of the task stack to 1024 bytes.

Note: The priority of task `IP_RxTask()` should be higher then the priority of an application task which uses the stack.

4.3.5 IP_Exec()

Description

Checks if the driver has received a packet and handles timers.

Prototype

```
void IP_Exec ( void );
```

Additional information

This function is normally called from an endless loop in `IP_Task()`. If no particular IP task is implemented in your project, `IP_Exec()` should be called regularly.

4.4 Network interface configuration and handling functions

4.4.1 IP_NI_ConfigPHYAddr()

Description

Configures the PHY address.

Prototype

```
void IP_NI_ConfigPHYAddr ( unsigned Unit,
                           U8       Addr );
```

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.
Addr	[IN] 5-bit address.

Table 4.30: IP_NI_ConfigPHYAddr() parameter list

Additional information

The PHY address is a 5-bit value. The available embOS/IP drivers try to detect the PHY address automatically, therefore this should not be called. If you use this function to set the address explicitly, the function must be called from within `IP_X_Config()`. Refer to *IP_X_Configure()* on page 256.

4.4.2 IP_NI_ConfigPHYMode()

Description

Configures the PHY mode.

Prototype

```
void IP_NI_ConfigPHYMode ( unsigned Unit,
                           U8       Mode );
```

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.
Mode	[IN] The operating mode of the PHY.

Table 4.31: IP_NI_ConfigPHYMode() parameter list

Valid values for parameter Mode

Value	Description
IP_PHY_MODE_MII	Phy uses the Media Independent Interface (MII).
IP_PHY_MODE_RMII	Phy uses the Reduced Media Independent Interface (RMII).

Additional information

The PHY can be connected to the MAC via two different modes, MII or RMII. Refer to section *MII / RMII: Interface between MAC and PHY* on page 23 for detailed information about the differences of the MII and RMII modes.

The selection which mode is used is normally done correctly by the hardware. The mode is typically sampled during power-on RESET. If you use this function to set the mode explicitly, the function must be called from within `IP_X_Config()`. Refer to *IP_X_Configure()* on page 256.

4.4.3 IP_NI_ConfigPoll()

Description

Select polled mode for the network interface. This should be used only if the network interface can not activate an ISR itself.

Prototype

```
void IP_NI_ConfigPoll( unsigned Unit );
```

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.

Table 4.32: IP_NI_ConfigPoll() parameter list

4.4.4 IP_NI_ForceCaps()

Description

Allows to force capabilities to be set for an interface. Typically this is used to allow the checksum calculation capabilities to be set manually. Typically this is used to give the target a performance boost in high traffic applications on stable networks, where the occurence of wrong checksums is unlikely.

Prototype

```
void IP_NI_ForceCaps( U8 IFace,
                      U8 CapsForcedMask,
                      U8 CapsForcedValue );
```

Parameter	Description
IFace	[IN] Zero-based index of available network interfaces.
CapsForcedMask	[IN] Capabilities mask. For a list of driver capabilities please refer to IP.h and look for the "Driver capabilities" section.
CapsForcedValue	[IN] Value mask for the capabilities to force.

Table 4.33: IP_NI_ConfigPoll() parameter list

Example

Forcing the capability bits 0 to value '0' and bit 2 to value '1' for the first interface can be done as shown in the code example below:

```
IP_NI_ForceCaps(0, 5, 4);
```


4.4.5 IP_NI_SetTxBufferSize()

Description

Sets the size of the Tx buffer of the network interface.

Prototype

```
int IP_NI_SetTxBufferSize ( unsigned Unit,
                            U8       NumBytes );
```

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.
NumBytes	[IN] Size of the Tx buffer (at least size of the MTU + 16 bytes for Ethernet.)

Table 4.34: IP_NI_SetTxBufferSize() parameter list

Return value

-1: Not supported by the network interface driver.
0: OK
1: Error, called after driver initialization has been completed.

Additional information

The default Tx buffer size is 1536 bytes. It can be useful to reduce the buffer size on systems with less RAM and an application that uses a small MTU. According to RFC 576 bytes is the smallest possible MTU. The size of the Tx buffer should be at least MTU + 16 bytes for Ethernet header and footer. The function should be called in IP_X_Config().

Note: This function is not implemented in all network interface drivers, since not all Media Access Controllers (MAC) support variable buffer sizes.

4.5 Other IP stack functions

4.5.1 IP_GetAddrMask()

Description

Returns the IP address and the subnet mask of the device in network byte order (for example, 192.168.1.1 is returned as 0xc0a80101).

Prototype

```
void IP_GetAddrMask ( U8 IFace, U32 * pAddr, U32 * pMask );
```

Parameter

Parameter	Description
IFace	[IN] Interface.
pAddr	[OUT] Address to store the IP address.
pMask	[OUT] Address to store the subnet mask.

Table 4.35: IP_GetAddrMask() parameter list

4.5.2 IP_GetCurrentLinkSpeed()

Description

Returns the current link speed of the first interface (interface ID ’0’).

Prototype

```
int IP_GetCurrentLinkSpeed( void );
```

Return value

0: link speed unknown
1: link speed is 10 Mbit/s
2: link speed is 100 Mbit/s
3: link speed is 1000 Mbit/s

Additional information

The application should check if the link is up before a packet will be sent. It can take 2-3 seconds till the link is up if the PHY has been reset.

Example

```
//
// Wait until link is up.
//
while (IP_GetCurrentLinkSpeed() == 0) {
  OS_IP_Delay(100);
}
```


4.5.3 IP_GetCurrentLinkSpeedEx()

Description

Returns the current link speed of the selected interface.

Prototype

```
int IP_GetCurrentLinkSpeedEx( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	[IN] Interface Id (zero-based).

Table 4.36: IP_GetCurrentLinkSpeedEx() parameter list

Return value

0: link speed unknown
1: link speed is 10 Mbit/s
2: link speed is 100 Mbit/s
3: link speed is 1000 Mbit/s

Additional information

The application should check if the link is up before a packet will be sent. It can take 2-3 seconds till the link is up if the PHY has been reset.

Example

```
//
// Wait until link is up.
//
while (IP_GetCurrentLinkSpeedEx(0) == 0) {
  OS_IP_Delay(100);
}
```


4.5.4 IP_GetGWAddr()

Description

Returns the gateway address of the interface in network byte order (for example, 192.168.1.1 is returned as 0xc0a80101).

Prototype

```
U32  IP_GetGWAddr ( U8 IFace );
```

Parameter

Parameter	Description
IFace	[IN] Number of interface.

Table 4.37: IP_GetGWAddr() parameter list

Return value

The gateway address of the interface.

4.5.5 IP_GetHWAddr()

Description

Returns the hardware address (Media Access Control address) of the interface.

Prototype

```
void IP_GetHWAddr ( U8 IFace, U8 * pDest, unsigned Len );
```

Parameter

Parameter	Description
IFace	[IN] Number of interface.
pDest	[OUT] Address of the buffer to store the 48-bit MAC address.
Len	[IN] Size of the buffer. Should be at least 6-bytes.

Table 4.38: IP_GetHWAddr() parameter list

4.5.6 IP_GetIPAddr()

Description

Returns the IP address of the interface.

Prototype

```
U32  IP_GetIPAddr( U8 IFace );
```

Parameter

Parameter	Description
IFace	[IN] Number of interface.

Table 4.39: IP_GetIPAddr() parameter list

Return value

The IP address of the interface in network byte order (for example, 192.168.1.1 is returned as 0xc0a80101).

Example

```
void PrintIFaceIPAddr(void) {
  char ac[16];
  U32 IPAddr;

  IPAddr = IP_GetIPAddr(0);
  IP_PrintIPAddr(ac, IPAddr, sizeof(ac));
  printf("IP Addr: %s\n", ac);
}
```


4.5.7 IP_GetIPPacketInfo()

Description

Returns the start address of the data part of an IP packet.

Prototype

```
const char * IP_GetIPPacketInfo( IP_PACKET * pPacket );
```

Parameter

Parameter	Description
pPacket	[IN] Pointer to an IP_PACKET structure.

Table 4.40: IP_GetIPPacketInfo() parameter list

Return value

0 > Start address of the data part of the IP packet.
0 On failure.

Example

```
/*********************************************************************
*
*       _pfOnRxICMP
*/
static int _pfOnRxICMP(IP_PACKET * pPacket) {
  const char * pData;

  pData = IP_GetIPPacketInfo(pPacket);
  if(*pData == 0x08) {
    printf("ICMP echo request received!\n");
  }
  if(*pData == 0x00) {
    printf("ICMP echo reply received!\n");
  }
  return 0;
}
```


4.5.8 IP_GetRawPacketInfo()

Description

Returns the start address of the raw data part of an IP packet.

Prototype

```
const char * IP_GetRawPacketInfo( IP_PACKET * pPacket,
                                  U16       * pNumBytes );
```

Parameter

Parameter	Description
pPacket	[IN] Pointer to an IP_PACKET structure.
pNumBytes	[OUT] Length of the packet.

Table 4.41: IP_GetRawPacketInfo() parameter list

Return value

0 > Start address of the raw data part of the IP packet.
0 On failure.

4.5.9 IP_GetVersion()

Description

Returns the version number of the stack.

Prototype

```
int IP_GetVersion ( void );
```

Additional information

The format of the version number: `<Major><Minor><Minor><Revision><Revision>`. For example, the return value `10201` means version 1.02a.

4.5.10 IP_ICMP_SetRxHook()

Description

Sets a hook function which will be called if target receives a ping packet.

Prototype

```
void IP_ICMP_SetRxHook(IP_RX_HOOK * pf);
```

Parameter

Parameter	Description
pf	Pointer to the callback function of type IP_RX_HOOK.

Table 4.42: IP_ICMP_SetRxHook() parameter list

Additional information

The return value of the callback function is relevant for the further processing of the ICMP packet. A return value of 0 indicates that the stack has to process the packet after the callback has returned. A return value of 1 indicates that the packet will be freed directly after the callback has returned.

The prototype for the callback function is defined as follows:

```
typedef int (IP_RX_HOOK)(IP_PACKET * pPacket);
```

Example

```
/*********************************************************************
*
*       Local defines, configurable
*
**********************************************************************
*/
#define HOST_TO_PING      0xC0A80101

/*********************************************************************
*
*       _pfOnRxICMP
*/
static int _pfOnRxICMP(IP_PACKET * pPacket) {
  const char * pData;

  pData = IP_GetIPPacketInfo(pPacket);
  if(*pData == 0x08) {
    printf("ICMP echo request received!\n");
  }
  if(*pData == 0x00) {
    printf("ICMP echo reply received!\n");
  }
  return 0; // Give packet back to the stack for further processing.
}

/*********************************************************************
*
*       PingTask
*/
void PingTask(void) {
  int Seq;
  char * s = "This is a ICMP echo request!";

  while (IP_IFaceIsReady() == 0) {
    OS_Delay(50);
  }
  IP_ICMP_SetRxHook(_pfOnRxICMP);
  Seq = 1111;
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
    IP_SendPing(htonl(HOST_TO_PING), s, strlen(s), Seq++);
  }
}
```


4.5.11 IP_IFaceIsReady()

Description

Checks if the interface is ready for usage. Ready for usage means that the target has a physical link detected and a valid IP address.

Prototype

```
int IP_IFaceIsReady ( void );
```

Return value

1 network interface is ready.
0 network interface is not ready.

Additional information

The application has to check if the link is up before a packet will be sent and if the interface is configured. If a DHCP server is used for configuring your target, this function has to be called to assure that no application data will be sent before the target is ready.

Example

```
//
// Wait until interface is ready.
//
while (IP_IFaceIsReady() == 0) {
  OS_Delay(100);
}
```


4.5.12 IP_IFaceIsReadyEx()

Description

Checks if the specified interface is ready for usage. Ready for usage means that the target has a physical link detected and a valid IP address.

Prototype

```
int IP_IFaceIsReadyEx ( unsigned IFaceId );
```

Return value

1 network interface is ready.
0 network interface is not ready.

Additional information

The application has to check if the link is up before a packet will be sent and if the interface is configured. If a DHCP server is used for configuring your target, this function has to be called to assure that no application data will be sent before the target is ready.

Example

```
//
// Wait until second interface is ready.
//
while (IP_IFaceIsReadyEx(1) == 0) {
  OS_Delay(100);
}
```


4.5.13 IP_PrintIPAddr()

Description

Convert an 4-byte IP address to a dots-and-number string.

Prototype

```
int IP_PrintIPAddr( char * pDest,
                    U32    IPAddr,
                    int    BufferSize );
```

Parameter

Parameter	Description
pDest	[OUT] Buffer to store the IP address string.
IPAddr	[IN] IP address in network byte order.
Buffersize	[IN] Size of buffer pDest. Should be 16 byte to store an IPv4 address.

Table 4.43: IP_PrintIPAddr() parameter list

Return value

0 on error. Size of the buffer is too small.
>0 on success. Length of the IP address string.

Example

```
void PrintIPAddr(void) {
  U32 IPAddr;
  char ac[16];

  IPAddr = 0xC0A80801;              // IP address: 192.168.8.1
  IP_PrintIPAddr(ac, IPAddr, sizeof(ac));
  printf("IP address: %s\n", ac);  // Output: IP address: 192.168.8.1
}
```


4.5.14 IP_SendPacket()

Description

Sends a user defined packet on the interface. The packet will not be modified by the stack. `IP_SendPacket()` allocates a packet control block (`IP_PACKET`) and adds it to the Out queue of the interface.

Prototype

```
int IP_SendPacket( unsigned IFace,
                   void   * pData,
                   int      NumBytes );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based interface index.
pData	[IN] Data packet that should be sent.
Numbytes	[IN] Length of data which should be sent.

Table 4.44: IP_SendPacket() parameter list

Return value

```
 0   O.K., packet in out queue
-1   Error: Could not allocate a packet control block
 1   Error: Interface can not send
```


4.5.15 IP_SendPing()

Description

Sends a single "ping" (ICMP echo request) to the specified host.

Prototype

```
int IP_SendPing ( ip_addr    host,
                  char     * data,
                  unsigned   datalen,
                  U16        pingseq );
```

Parameter

Parameter	Description
host	[IN] 4-byte IPv4 address in network endian byte order.
data	[IN] Ping data, NULL if do not care.
datalen	[IN] Length of data to attach to ping request.
pingseq	[IN] Ping sequence number.

Table 4.45: IP_SendPing() parameter list

Return value

Returns 0 if ICMP echo request was successfully sent, else negative error message.

Additional information

If you call this function with activated logging, the ICMP reply or (in case of an error) the error message will be sent to stdout. To enable the output of ICMP status messages, add the message type IP_MTYPE_ICMP to the log filter and the warn filter. Refer to *Debugging* on page 433 for detailed information about logging.

4.5.16 IP_SendPingEx()

Description

Sends a single "ping" (ICMP echo request) to the specified host using the selected interface.

Prototype

```
int IP_SendPingEx ( U32         IFaceId,
                    ip_addr     host,
                    char      * data,
                    unsigned    datalen,
                    U16         pingseq );
```

Parameter

Parameter	Description
IFaceId	[IN] Interface index (zero-based).
host	[IN] 4-byte IPv4 address in network endian byte order.
data	[IN] Ping data, NULL if do not care.
datalen	[IN] Length of data to attach to ping request.
pingseq	[IN] Ping sequence number.

Table 4.46: IP_SendPingEx() parameter list

Return value

Returns 0 if ICMP echo request was successfully sent, else negative error message.

Additional information

If you call this function with activated logging, the ICMP reply or (in case of an error) the error message will be sent to stdout. To enable the output of ICMP status messages, add the message type IP_MTYPE_ICMP to the log filter and the warn filter. Refer to *Debugging* on page 433 for detailed information about logging.

4.5.17 IP_SetRxHook()

Description

Sets a hook function which will be called if target receives a packet.

Prototype

```
void IP_SetRxHook(IP_RX_HOOK * pf);
```

Parameter

Parameter	Description
pf	Pointer to the callback function of type IP_RX_HOOK.

Table 4.47: IP_SetRxHook() parameter list

Additional information

The return value of the callback function is relevant for the further processing of the packet. A return value of 0 indicates that the stack has to process the packet after the callback has returned. A return value of >0 indicates that the packet will be freed directly after the callback has returned.

The prototype for the callback function is defined as follows:

```
typedef int (IP_RX_HOOK)(IP_PACKET * pPacket);
```

Example

Refer to *IP_ICMP_SetRxHook()* on page 99 for an example.

4.6 Stack internal functions, variables and data-structures

embOS/IP internal functions, variables and data-structures are not explained here as they are in no way required to use embOS/IP. Your application should not rely on any of the internal elements, as only the documented API functions are guaranteed to remain unchanged in future versions of embOS/IP.

Chapter 5

Socket interface

The embOS/IP socket API is almost compatible to the Berkeley socket interface. The Berkeley socket interface is the de facto standard for socket communication. All API functions are described in this chapter.

5.1 API functions

The table below lists the available socket API functions.

Function	Description
Socket interface	
accept()	Accepts an incoming attempt on a socket.
bind()	Assigns a name to an unnamed socket.
closesocket()	Closes an existing socket.
connect()	Establishes a connection to a socket.
gethostbyname()	Resolves a host name into an IP address.
getpeername()	Returns the IP addressing information of the connected host.
getsockname()	Returns the current name for the specified socket.
getsockopt()	Returns the socket options.
listen()	Marks a socket as accepting connections.
recv()	Receives data from a connected socket.
recvfrom()	Receives a datagram and stores the source address.
select()	Checks if socket is ready.
send()	Sends data on a connected socket.
sendto()	Sends data to a specified address.
setsockopt()	Sets a socket option.
shutdown()	Disables sends or receives on a socket.
socket()	Creates an unbound socket.
Helper macros	
ntohl	Converts a unsigned long value from network to host byte order.
htonl	Converts a unsigned long value from host byte order to network byte order.
htons	Converts a unsigned short value from host byte order to network byte order.
ntohs	Converts a unsigned short value from network to host byte order.

Table 5.1: embOS/IP socket API function overview

5.1.1 accept()

Description

Accepts an incoming attempt on a socket.

Prototype

```
long accept ( long               Socket,
              struct sockaddr  * pAddr,
              int              * pAddrLen );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
pAddr	[OUT] An optional pointer to a buffer where the address of the connecting entity should be stored. The format of the address depends on the defined address family which was defined when the socket was created.
pAddrLen	[OUT] An optional pointer to an integer where the length of the received address should be stored. Just like the format of the address, the length of the address depends on the defined address family.

Table 5.2: accept() parameter list

Return value

The returned value is a handle for the socket on which the actual connection will be made.

-1 in case of an error.

Additional information

This call is used with connection-based socket types, currently with SOCK_STREAM. Refer to *socket()* on page 135 for more information about the different socket types.

Before calling accept(), the used socket Socket has to be bound to an address with bind() and should be listening for connections after calling listen(). accept() extracts the first connection on the queue of pending connections, creates a new socket with the same properties of Socket and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, accept() blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept() returns and reports an error. The accepted socket is used to read and write data to and from the socket which is connected to this one; it is not used to accept more connections. The original socket Socket remains open for accepting further connections.

The argument pAddr is a result parameter that is filled in with the address of the connecting entity as known to the communications layer. The exact format of the pAddr parameter is determined by the domain in which the communication is occurring. The pAddrLen is a value-result parameter. It should initially contain the amount of space pointed to by pAddr.

5.1.2 bind()

Description

Assigns a name (port) to an unnamed socket.

Prototype

```
int bind ( long              Socket,
           struct sockaddr * pAddr,
           int               AddrLen );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
pAddr	[IN] A pointer to a buffer where the address of the connecting entity is stored. The format of the address depends on the defined address family which was defined when the socket was created.
AddrLen	[IN] The length of the address.

Table 5.3: bind() parameter list

Return value

0 on success.
-1 on failure.

Additional information

When a socket is created with `socket()` it exists in a name space (address family) but has no name assigned. `bind()` is used on an unconnected socket before subsequent calls to the `connect()` or `listen()` functions. `bind()` assigns the name pointed to by `pAddr` to the socket.

5.1.3 closesocket()

Description

Closes an existing socket.

Prototype

```
int closesocket ( long Socket );
```

Parameter

Parameter	Description
Socket	[IN] Socket descriptor of the socket that should be closed.

Table 5.4: closesocket() parameter list

Return value

0 on success.
-1 on failure.

Additional information

`closesocket()` closes a connection on the socket associated with `Socket` and the socket descriptor associated with `Socket` will be returned to the free socket descriptor pool. Once a socket is closed, no further socket calls should be made with it.

If the socket promises reliable delivery of data and `SO_LINGER` is set, the system will block the caller on the `closesocket()` attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the `setsockopt()` call when `SO_LINGER` is requested). If `SO_LINGER` is disabled and a `closesocket()` is issued, the system will process the close in a manner that allows the caller to continue as quickly as possible. If `SO_LINGER` is enabled with a timeout period of '0' and a `closesocket()` is issued, the system will perform a hard close.

Example

```
/*********************************************************************
*
*       _CloseSocketGracefully()
*
*  Function description
*    Wrapper for closesocket() with linger enabled to verify a gracefully
*    disconnect.
*/
static int _CloseSocketGracefully(long pConnectionInfo) {
  struct linger Linger;

  Linger.l_onoff  = 1;  // Enable linger for this socket.
  Linger.l_linger = 1;  // Linger timeout in seconds
  setsockopt(hSocket, SOL_SOCKET, SO_LINGER, &Linger, sizeof(Linger));
  return closesocket(hSocket);
}

/*********************************************************************
*
*       _CloseSocketHard()
*
*  Function description
*    Wrapper for closesocket() with linger option enabled to perform a hard close.
*/
static int _CloseSocketHard(long hSocket) {
  struct linger Linger;

  Linger.l_onoff  = 1;  // Enable linger for this socket.
  Linger.l_linger = 0;  // Linger timeout in seconds
  setsockopt(hSocket, SOL_SOCKET, SO_LINGER, &Linger, sizeof(Linger));
  return closesocket(hSocket);
}
```


5.1.4 connect()

Description

Establishes a connection to a socket.

Prototype

```
int connect ( long               Socket,
              struct sockaddr *  pAddr,
              int                AddrLen );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying an unconnected socket.
pAddr	[IN] A pointer to a buffer where the address of the connecting entity is stored. The format of the address depends on the defined address family which was defined when the socket was created.
AddrLen	[IN] A pointer to an integer where the length of the received address is stored. Just like the format of the address, the length of the address depends on the defined address family.

Table 5.5: connect() parameter list

Return value

0 on success.
-1 on failure.

Additional information

If Socket is of type SOCK_DGRAM, then this call specifies the peer with which the socket is to be associated. pAddr defines the address to which datagrams are sent and the only address from which datagrams are received.

If Socket is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by pAddr which is an address in the communications space of the socket. Each communications space interprets the pAddr parameter in its own way.

Generally, stream sockets may successfully connect() only once; datagram sockets may use connect() multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a NULL address.

If a connect is in progress and the socket is blocking, the connect call waits until connected or an error to happen. If the socket is non-blocking (refer to *setsockopt()* on page 132 for more information), 0 is returned.

You can use the getsockopt() function (refer to *getsockopt()* on page 120) to determine the status of the connect attempt.

Example

```
#define SERVER_PORT             1234
#define SERVER_IP_ADDR          0xC0A80101      // 192.168.1.1

/*********************************************************************
*
*       _TCPClientTask
*
*  Function description
*    Creates a connection to a given IP address, TCP port.
*/
static void _TCPClientTask(void) {
  int               TCPSockID;
  struct sockaddr_in ServerAddr;
```



```
  int                ConnectStatus;

  //
  // Wait until link is up. This can take 2-3 seconds if PHY has been reset.
  //
  while (IP_GetCurrentLinkSpeed() == 0) {
    OS_Delay(100);
  }

  while(1) {
    TCPSockID = socket(AF_INET, SOCK_STREAM, 0);  // Open socket
    if (TCPSockID < 0) {                          // Error, Could not get socket
      while (1) {
        OS_Delay(20);
      }
    } else {
      //
      // Connect to server
      //
      ServerAddr.sin_family      = AF_INET;
      ServerAddr.sin_port        = htons(SERVER_PORT);
      ServerAddr.sin_addr.s_addr = htonl(SERVER_IP_ADDR);
      ConnectStatus              = connect(TCPSockID,
                                           (struct sockaddr *)&ServerAddr,
                                            sizeof(struct sockaddr_in));
      if (ConnectStatus == 0) {
        //
        // Do something...
        //
      }
    }
    closesocket(TCPSockID);
    OS_Delay(50);
  }
}
```


5.1.5 gethostbyname()

Description

Resolve a host name into an IP address.

Prototype

```
struct hostent * gethostbyname (char * name);
```

Parameter

Parameter	Description
name	[OUT] Host name.

Table 5.6: gethostbyname() parameter list

Return value

On success, a pointer to a `hostent` structure is returned. Refer to *Structure hostent* on page 140 for detailed information about the `hostent` structure.
On failure, it returns NULL.

Additional information

The function is called with a string containing the host name to be resolved as a fully-qualified domain name (for example, myhost.mydomain.com).

Example

```
static void _DNSClient() {
  struct hostent *pHostEnt;
  char **ps;
  char **ppAddr;
  //
  // Wait until link is up.
  //
  while (IP_IFaceIsReady() == 0) {
    OS_Delay(100);
  }
  while(1) {
    pHostEnt = gethostbyname("www.segger.com");
    if (pHostEnt == NULL) {
      printf("Could not resolve host addr.\n");
      break;
    }
    printf("h_name: %s\n", pHostEnt->h_name);
    //
    // Show aliases
    //
    ps = pHostEnt->h_aliases;
    for (;;) {
      char * s;
      s = *ps++;
      if (s == NULL) {
        break;
      }
      printf("h_aliases: %s\n", s);
    }
    //
    // Show IP addresses
    //
    ppAddr = pHostEnt->h_addr_list;
    for (;;) {
      U32 IPAddr;
      char * pAddr;
      char ac[16];

      pAddr = *ppAddr++;
      if (pAddr == NULL) {
        break;
      }
```



```
      IPAddr = *(U32*)pAddr;
      IP_PrintIPAddr(ac, IPAddr, sizeof(ac));
      printf("IP Addr: %s\n", ac);
    }
  }
}
```


5.1.6 getpeername()

Description

Fills the passed structure `sockaddr` with the IP addressing information of the connected host.

Prototype

```
int getpeername ( long              Socket,
                  struct sockaddr * pAddr,
                  struct int      * pAddrLen );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
pAddr	[OUT] A pointer to a structure of type `sockaddr` in which the IP address information of the connected host should be stored.
pAddrLen	[OUT] A pointer to an integer to store the length of socket address.

Table 5.7: getpeername() parameter list

Return value

0 on success.
-1 on failure.

Additional information

Refer to *Structure sockaddr* on page 137 for detailed information about the structure `sockaddr`.

5.1.7 getsockname()

Description

Returns the current name for the specified socket.

Prototype

```
int getsockname ( long              Socket,
                  struct sockaddr * pAddr );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
pAddr	[OUT] A pointer to a structure of type sockaddr in which the IP address information of the connected host should be stored.

Table 5.8: getsockname() parameter list

Return value

0 on success.
-1 on failure.

Additional information

Refer to *Structure sockaddr* on page 137 for detailed information about the structure sockaddr.

5.1.8 getsockopt()

Description

Returns the options associated with a socket.

Prototype

```
int getsockopt ( long   Socket,
                 int    Level,
                 int    Option,
                 void * pData,
                 int    DataLen );
```


Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
Level	[IN] Compatibility parameter for setsockopt() and getsockopt(). Use symbol SOL_SOCKET.
Option	[IN] The socket option which should be retrieved.
pData	[OUT] A pointer to the buffer in which the value of the requested option should be stored.
DataLen	[IN] The size of the data buffer.

Table 5.9: getsockopt() parameter list

Valid values for parameter Option

Value	Description
Standard option flags.	
SO_ACCEPTCONN	Indicates that socket is in listen mode.
SO_DONTROUTE	Indicates that outgoing messages must bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_KEEPALIVE	Indicates that the periodic transmission of messages on a connected socket is enabled. If the connected party fails to respond to these messages, the connection is considered broken.
SO_LINGER	Indicates that linger on close is enabled.
SO_NOSLOWSTART	Indicates that suppress slow start on this socket is enabled.
SO_TIMESTAMP	Indicates that the TCP timestamp option is enabled.
embOS/IP socket options.	
SO_ERROR	Stores the latest socket error in pData and clears the error in socket structure.
SO_MYADDR	Stores the IP address of the used interface in pData.
SO_RCVTIMEO	Returns the timeout for recv(). A return value of 0 indicates that no timeout is set.
SO_NONBLOCK	Gets sockets blocking status. Allows the caller to specify blocking or non-blocking IO that works the same as the other Boolean socket options. pData points to an integer value which will contain a non-zero value to set non-blocking IO or a 0 value to reset non-blocking IO.

Return value

0 on success.
-1 on failure.

Additional information

getsockopt() retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in pData. Options can exist at multiple protocol levels, but they are always present at the uppermost “socket” level. Options affect socket operations, such as the packet routing.

The value associated with the selected option is returned in the buffer pData. The integer pointed to by DataLen should originally contain the size of this buffer; on return, it will be set to the size of the value returned. For SO_LINGER, this will be the size of a LINGER structure. For most other options, it will be the size of an integer.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specified. If the option was never set with setsockopt(), then getsockopt() returns the default value for the option.

The option SO_ERROR returns 0 or the number of the socket error and clears the socket error. The following table lists the socket errors.

Symbolic name	Value	Description
IP_ERR_SEND_PENDING	1	Packet to send is not sent yet.
IP_ERR_MISC	-1	Miscellaneous errors that do not have a specific error code.
IP_ERR_TIMEDOUT	-2	Operation timed out.
IP_ERR_ISCONN	-3	Socket is already connected.
IP_ERR_OP_NOT_SUPP	-4	Operation not supported for selected socket.
IP_ERR_CONN_ABORTED	-5	Connection was aborted.
IP_ERR_WOULD_BLOCK	-6	Socket is in non-blocking state and the current operation would block the socket if not in non-blocking state.
IP_ERR_CONN_REFUSED	-7	Connection refused by peer.
IP_ERR_CONN_RESET	-8	Connection has been reset.
IP_ERR_NOT_CONN	-9	Socket is not connected.
IP_ERR_ALREADY	-10	Socket already is in the requested state.
IP_ERR_IN_VAL	-11	Passed value for configuration is not valid.
IP_ERR_MSG_SIZE	-12	Message is too big to send.
IP_ERR_PIPE	-13	Socket is not in the correct state for this operation.
IP_ERR_DEST_ADDR_REQ	-14	Destination addr. has not been specified.
IP_ERR_SHUTDOWN	-15	Connection has been closed as soon as all data has been received upon a FIN request.
IP_ERR_NO_PROTO_OPT	-16	Unknown socket option for setsockopt() or getsockopt().
IP_ERR_NO_MEM	-18	Not enough memory in the memory pool.
IP_ERR_ADDR_NOT_AVAIL	-19	No known path to send to the specified addr.
IP_ERR_ADDR_IN_USE	-20	Socket already has a connection to this addr. and port or is already bound to this addr.
IP_ERR_IN_PROGRESS	-22	Operation is still in progress.
IP_ERR_NO_BUF	-23	No internal buffer was available.
IP_ERR_NOT_SOCK	-24	Socket has not been opened or has already been closed
IP_ERR_FAULT	-25	Generic error for a failed operation.
IP_ERR_NET_UNREACH	-26	No path to the desired network available.
IP_ERR_PARAM	-27	Invalid parameter to function.
IP_ERR_LOGIC	-28	Logical error that should not have happened.
IP_ERR_NOMEM	-29	System error: No memory for requested operation.

Table 5.10: embOS/IP socket error types

Symbolic name	Value	Description
IP_ERR_NOBUFFER	-30	System error: No internal buffer available for the requested operation.
IP_ERR_RESOURCE	-31	System error: Not enough free resources available for the requested operation.
IP_ERR_BAD_STATE	-32	Socket is in an unexpected state.
IP_ERR_TIMEOUT	-33	Requested operation timed out.
IP_ERR_NO_ROUTE	-36	Net error: Destination is unreachable.

Table 5.10: embOS/IP socket error types

5.1.9 listen()

Description

Prepares the socket to accept connections.

Prototype

```
int listen ( long Socket,
             int  Backlog );
```

Parameter

Parameter	Description
Socket	[IN] Socket descriptor of an unconnected socket.
Backlog	[IN] Backlog for incoming connections. Defines the maximum length of the queue of pending connections.

Table 5.11: listen() parameter list

Return value

On success 0.
On failure, it returns -1.

Additional information

The `listen()` call applies only to sockets of type `SOCK_STREAM`. If a connection request arrives when the queue is full, the client will receive an error with an indication of `ECONNREFUSED`.

Example

```
/*********************************************************************
*
*       _ListenAtTcpAddr
*
*  Function description
*    Starts listening at the given TCP port.
*/
static int _ListenAtTcpAddr(U16 Port) {
  int                Sock;
  struct sockaddr_in Addr;

  Sock = socket(AF_INET, SOCK_STREAM, 0);
  memset(&Addr, 0, sizeof(Addr));
  Addr.sin_family      = AF_INET;
  Addr.sin_port        = htons(Port);
  Addr.sin_addr.s_addr = INADDR_ANY;
  bind(Sock, (struct sockaddr *)&Addr, sizeof(Addr));
  listen(Sock, 1);
  return Sock;
}
```


5.1.10 recv()

Description

Receives data from a connected socket.

Prototype

```
int recv ( long   Socket,
           char * pRecv,
           int    Length,
           int    Flags );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
pRecv	[OUT] A pointer to a buffer for incoming data.
Length	[IN] The length of buffer pRecv in bytes.
Flags	[IN] OR-combination of one or more of the following valid values.

Table 5.12: recv() parameter list

Valid values for parameter Flag

Value	Description
MSG_PEEK	"Peek" at the data present on the socket; the data are returned, but not consumed, so that a subsequent receive operation will see the same data.

Return value

If no error occurs, recv() returns the number of bytes received. If the connection has been gracefully closed, the return value is zero. Otherwise, -1 is returned, and a specific error code can be retrieved by calling getsockopt(). Refer to *getsockopt()* on page 120 for detailed information.

Additional information

If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from. Refer to *connect()* on page 114 for more information about the different types of sockets.

You can only use the recv() function on a connected socket. To receive data on a socket, whether it is in a connected state or not refer to *recvfrom()* on page 126.

If no messages are available at the socket and the socket is blocking, the receive call waits for a message to arrive. If the socket is non-blocking (refer to *setsockopt()* on page 132 for more information), –1 is returned.

You can use the select() function to determine when more data arrives.

5.1.11 recvfrom()

Description

Receives a datagram and stores the source address.

Prototype

```
int recvfrom ( long               Socket,
               char             * pRecv,
               int                Length,
               int                Flags,
               struct sockaddr  * pAddr,
               int              * pAddrLen );
```

Parameter

Parameter	Description
Socket	[IN] A socket descriptor of a socket.
pRecv	[OUT] A pointer to a buffer for incoming data.
Length	[IN] Specifies the size of the buffer pRecv in bytes.
Flags	[IN] OR-combination of one or more of the values listed in the table below.
pAddr	[OUT] An optional pointer to a buffer where the address of the connecting entity is stored. The format of the address depends on the defined address family which was defined when the socket was created.
pAddrLen	[IN/OUT] An optional pointer to an integer where the length of the received address is stored. Just like the format of the address, the length of the address depends on the defined address family.

Table 5.13: recvfrom() parameter list

Valid values for parameter Flags

Value	Description
MSG_PEEK	"Peek" at the data present on the socket; the data are returned, but not consumed, so that a subsequent receive operation will see the same data.

Return value

The number of bytes received or -1 if an error occurred.

Additional information

If pAddr is not a NULL pointer, the source address of the message is filled in. pAddrLen is a value-result parameter, initialized to the size of the buffer associated with pAddr, and modified on return to indicate the actual size of the address stored there.

If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from. Refer to *socket()* on page 135 for more information about the different types of sockets.

If no messages are available at the socket and the socket is blocking, the receive call waits for a message to arrive. If the socket is non-blocking (refer to *setsockopt()* on page 132 for more information), –1 is returned.

You can use the select() function to determine when more data arrives.

5.1.12 select()

Description

Examines the socket descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if some of their descriptors are ready for reading, ready for writing or have an exception condition pending.

Prototype

```
int select ( IP_FD_set * readfds,
             IP_FD_set * writefds,
             IP_FD_set * exceptfds;
             long        tv );
```

Parameter

Parameter	Description
readfds writefds exceptfds tv	See below.

Table 5.14: select() parameter list

Return value

Returns a non-negative value on success. A positive value indicates the number of ready descriptors in the descriptor sets. 0 indicates that the time limit specified by tv expired. On failure, select() returns -1 and the descriptor sets are not changed.

Additional information

On return, select() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned. Any of readfds, writefds, and exceptfds may be given as NULL pointers if no descriptors are of interest. Selecting true for reading on a socket descriptor upon which a listen() call has been performed indicates that a subsequent accept() call on that descriptor will not block.

In the standard Berkeley UNIX Sockets API, the descriptor sets are stored as bit fields in arrays of integers. This works in the UNIX environment because under UNIX socket descriptors are file system descriptors which are guaranteed to be small integers that can be used as indexes into the bit fields. In embOS/IP, socket descriptors are pointers and thus a bit field representation of the descriptor sets is not feasible. Because of this, the embOS/IP API differs from the Berkeley standard in that the descriptor sets are represented as instances of the following structure:

```
typedef struct IP_FD_SET {       // The select socket array manager
  unsigned fd_count;             // how many are SET?
  long fd_array[FD_SETSIZE];     // an array of SOCKETs
} IP_fd_set;
```

Instead of a socket descriptor being represented in a descriptor set via an indexed bit, an embOS/IP socket descriptor is represented in a descriptor set by its presence in the fd_array field of the associated IP_FD_SET structure. Despite this non-standard representation of the descriptor sets themselves, the following standard entry points are provided for manipulating such descriptor sets: IP_FD_ZERO (&fdset) initializes a descriptor set fdset to the null set. IP_FD_SET(fd, &fdset) includes a particular descriptor, fd, in fdset. IP_FD_CLR(fd, &fdset) removes fd from fdset. IP_FD_ISSET(fd, &fdset) is nonzero if fd is a member of fdset, zero otherwise. These entry points behave according to the standard Berkeley semantics.

You should be aware that the value of FD_SETSIZE defines the maximum number of descriptors that can be represented in a single descriptor set. The default value of FD_SETSIZE is 12. This value can be increased in the source code version of embOS/IP to accommodate a larger maximum number of descriptors at the cost of increased processor stack usage.

Another difference between the Berkeley and embOS/IP select() calls is the representation of the timeout parameter. Under Berkeley Sockets, the timeout parameter is represented by a pointer to a structure. Under embOS/IP sockets, a timeout is specified by the tv parameter, which defines the maximum number of seconds that should elapse before the call to select() returns. A tv parameter equal to 0 implies that select() should return immediately (effectively a poll of the sockets in the descriptor sets). A tv parameter equal to -1 implies that select() blocks forever unless one of its descriptors becomes ready.

The final difference between the Berkeley and embOS/IP versions of select() is the absence in the embOS/IP version of the Berkeley width parameter. The width parameter is of use only when descriptor sets are represented as bit arrays and was thus deleted in the embOS/IP implementation.

Note: Under rare circumstances, select() may indicate that a descriptor is ready for writing when in fact an attempt to write would block. This can happen if system resources necessary for a write are exhausted or otherwise unavailable. If an application deems it critical that writes to a file descriptor not block, it should set the descriptor for non-blocking I/O. Refer to *setsockopt()* on page 132 for detailed information.

Example

```
static void _Client() {
  long               Socket;
  struct sockaddr_in Addr;
  IP_fd_set          readfds;
  char               RecvBuffer[1472]
  int                r;

  while (IP_IFaceIsReady() == 0) {
    OS_Delay(100);
  }

Restart:
  Socket = socket(AF_INET, SOCK_DGRAM, 0);   // Open socket
  Addr.sin_family      = AF_INET;
  Addr.sin_port        = htons(2222);
  Addr.sin_addr.s_addr = INADDR_ANY;
  r = bind(Socket, (struct sockaddr *)&Addr, sizeof(Addr));
  if (r == -1){
    socketclose(Socket);
    OS_Delay(1000);
    goto Restart;
  }
  while(1) {
    IP_FD_ZERO(&readfds);                    // Clear the set
    IP_FD_SET(Socket, &readfds);             // Add descriptor to the set
    r = select(&readfds, NULL, NULL, 5000); // Check for activity.
    if (r <= 0) {
      continue;                              // No socket activity or error detected
    }
    if (IP_FD_ISSET(Socket, &readfds)) {
      IP_FD_CLR(Socket, &readfds);           // Remove socket from set
      r = recvfrom(Socket, RecvBuffer, sizeof(RecvBuffer), 0, NULL, NULL);
      if (r == -1){
        socketclose(Socket)
        goto Restart;
      }
    }
```



```
    OS_Delay(100);
  }
}
```


5.1.13 send()

Description

Sends data to a connected socket.

Prototype

```
int send ( long   Socket,
           char * pSend,
           int    Length,
           int    Flags );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
pSend	[IN] A pointer to a buffer of data which should be sent.
Length	[IN] The length of the message which should be sent.
Flags	[IN] OR-combination of one or more of the valid values listed in the table below.

Table 5.15: send() parameter list

Valid values for parameter Flags

Value	Description
MSG_DONTROUTE	Specifies that the data should not be subject to routing.

Return value

The total number of bytes which were sent or -1 if an error occurred.

Additional information

`send()` may be used only when the socket is in a connected state. Refer to *sendto()* on page 131 for information about sending data to a non-connected socket.

If no messages space is available at the socket to hold the message to be transmitted, then `send()` normally blocks, unless the socket has been placed in non-blocking I/O mode.

`MSG_DONTROUTE` is usually used only by diagnostic or routing programs.

5.1.14 sendto()

Description

Sends data to a specified address.

Prototype

```
int sendto ( long              Socket,
             char            * pSend,
             int               Length,
             int               Flags,
             struct sockaddr * pAddr,
             int               ToLen );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
pSend	[IN] A pointer to a buffer of data which should be sent.
Length	[IN] The length of the message which should be sent.
Flags	[IN] OR-combination of one or more of the valid values listed in the table below.
pAddr	[IN] An optional pointer to a buffer where the address of the connected entity is stored. The format of the address depends on the defined address family which was defined when the socket was created.
ToLen	[IN] The size of the address in pAddr.

Table 5.16: sendto() parameter list

Valid values for parameter Flags

Value	Description
MSG_DONTROUTE	Specifies that the data should not be subject to routing.

Return value

The total number of bytes which were sent or -1 if an error occurred.

Additional information

In contrast to send(), sendto() can be used at any time. The connection state is in which case the address of the target is given by the pAddr parameter.

5.1.15 setsockopt()

Description

Sets a socket option.

Prototype

```
int setsockopt ( long   Socket,
                 int    Level,
                 int    Option,
                 void * pData,
                 int    DataLen );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
Level	[IN] Compatibility parameter for setsockopt() and getsockopt(). Use symbol SOL_SOCKET.
Option	[IN] The socket option for which the value is to be set.
pData	[IN] A pointer to the buffer in which the value for the requested option is supplied.
DataLen	[IN] The size of the pData buffer.

Table 5.17: setsocketopt() parameter list

Valid values for parameter Option

Value	Description
Standard option flags.	
SO_DONTROUTE	Outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address. By default, this socket option is disabled.
SO_KEEPALIVE	Enable periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken. By default, this socket option is enabled.
SO_LINGER	Controls the action taken when unsent messages are queued on a socket and a closesocket() is performed. Refer to *closesocket()* on page 113 for detailed information about the linger option. By default, this socket option is disabled.
SO_TIMESTAMP	Enable the TCP timestamp option. By default, this socket option is disabled.
embOS/IP socket options.	
SO_CALLBACK	Sets zero-copy callback routine. Refer to *TCP zero-copy interface* on page 143 for detailed information.
SO_RCVTIMEO	Sets a timeout for recv(). This changes the behavior of recv(). recv() is by default a blocking function which only returns if data has been received. If a timeout is set recv() will return in case of data reception or timeout. By default, this socket option is disabled.

Value	Description
SO_NONBLOCK	Sets socket blocking status. Allows the caller to specify blocking or non-blocking IO that works the same as the other Boolean socket options. pData points to an integer value which will contain a non-zero value to set non-blocking IO or a 0 value to reset non-blocking IO. By default, this socket option is disabled.

Return value

0 on success

Example

```
void _EnableKeepAlive(long sock) {
    int v = 1;

    setsockopt(sock, SOL_SOCKET, SO_KEEPALIVE, &v, sizeof(v));
}
```


5.1.16 shutdown()

Description

Disables sends or receives on a socket.

Prototype

```
int shutdown( long Socket,
              int  Mode );
```

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
Mode	[IN] Indicator which part of communication should be disabled. Refer to additional information below.

Table 5.18: shutdown() parameter list

Return value

Returns 0 on success.
On failure, it returns -1.

Additional information

A `shutdown()` call causes all or part of a full-duplex connection on the socket associated with `Socket` to be shut down. If `Mode` is 0, then further receives will be disallowed. If `Mode` is 1, then further sends will be disallowed. If `Mode` is 2, then further sends and receives will be disallowed. The shutdown function does not block regardless of the `SO_LINGER` setting on the socket.

5.1.17 socket()

Description

Creates a socket. A socket is an endpoint for communication.

Prototype

```
long  socket ( int Domain,
               int Type,
               int Proto );
```

Parameter

Parameter	Description
Domain	[IN] Protocol family which should be used.
Type	[IN] Specifies the type of the socket.
Proto	[IN] Specifies the protocol which should be used with the socket. Must be set to zero.

Table 5.19: socket() parameter list

Valid values for parameter Domain

Value	Description
AF_INET	IPv4 - Internet protocol version 4

Valid values for parameter Type

Value	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket

Return value

A non-negative descriptor on success.
On failure, it returns -1.

Additional information

The Domain parameter specifies a communication domain within which communication will take place; the communication domain selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket.

A SOCK_STREAM socket provides sequenced, reliable, two-way connection based byte streams. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed - typically small - maximum length).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to UNIX pipes. A stream socket must be in a connected state before it can send or receive data.

A connection to another socket is created with a connect() call. Once connected, data can be transferred using send() and recv() calls. When a session has been completed, a closesocket() should be performed.

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data (for which the peer protocol has buffer space) cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will return -1 which indicates an error. The protocols optionally keep sockets “warm” by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (such as five minutes).

`SOCK_DGRAM` sockets allow sending of datagrams to correspondents named in `sendto()` calls. Datagrams are generally received with `recvfrom()`, which returns the next datagram with its return address.

The operation of sockets is controlled by socket-level options. The `getsockopt()` and `setsockopt()` functions are used to get and set options. Refer to *getsockopt()* on page 120 and *setsockopt()* on page 132 for detailed information.

5.2 Socket data structures

5.2.1 Structure sockaddr

Description

This structure holds socket address information for many types of sockets.

Prototype

```
struct sockaddr {
  U16     sa_family;
  char    sa_data[14];
};
```

Member	Description
sa_family	Address family. Normally AF_INET.
sa_data	The character array sa_data contains the destination address and port number for the socket.

Table 5.20: Structure sockaddr member list

Additional information

The structure sockaddr is mostly used as function parameter. To deal with struct sockaddr, a parallel structure struct sockaddr_in is implemented. The structure sockaddr_in is the same size as structure sockaddr, so that a pointer can freely be casted from one type to the other. Refer to *Structure sockaddr_in* on page 138 for more information and an example.

5.2.2 Structure sockaddr_in

Description

Structure for handling internet addresses.

Prototype

```
struct sockaddr_in {
  short          sin_family;
  unsigned short sin_port;
  struct in_addr sin_addr;
  char           sin_zero[8];
};
```

Member	Description
sin_family	Address family. Normally AF_INET.
sin_port	Port number for the socket.
sin_addr	Structure of type in_addr. The structure represents a 4-byte number that represents one digit in an IP address per byte.
sin_zero	sin_zero member is unused.

Table 5.21: Structure sockaddr_in member list

Example

Refer to *connect()* on page 114 for an example.

5.2.3 Structure in_addr

Description

4-byte number that represents one digit in an IP address per byte.

Prototype

```
struct in_addr {
  unsigned long  s_addr;
};
```

Member	Description
s_addr	Number that represents one digit in an IP address per byte.

Table 5.22: Structure in_addr member list

5.2.4 Structure hostent

Description

The hostent structure is used by functions to store information about a given host, such as host name, IPv4 address, and so on.

Prototype

```
struct hostent {
   char *   h_name;
   char **  h_aliases;
   int      h_addrtype;
   int      h_length;
   char **  h_addr_list;
};
```

Member	Description
h_name	Official name of the host.
h_aliases	Alias list.
s_addrtype	Host address type.
h_length	Length of the address.
s_addr_list	List of addresses from the name server.

Table 5.23: Structure in_addr member list

5.3 Error codes

The following table contains a list of generic error codes, generally full success is 0. Definite errors are negative numbers, and indeterminate conditions are positive numbers.

Symbolic name	Value	Description
Programming errors		
IP_ERR_PARAM	-10	Bad parameter.
IP_ERR_LOGIC	-11	Sequence of events that shouldn't happen.
System errors		
IP_ERR_NOMEM	-20	malloc() or calloc() failed.
IP_ERR_NOBUFFER	-21	Run out of free packets.
IP_ERR_RESOURCE	-22	Run out of other queue-able resource.
IP_ERR_BAD_STATE	-23	TCP layer error.
IP_ERR_TIMEOUT	-24	Timeout error on TCP layer.
Networking errors		
IP_ERR_BAD_HEADER	-32	Bad header at upper layer (for upcalls).
IP_ERR_NO_ROUTE	-33	Can not find a reasonable next IP hop.
Networking errors		
IP_ERR_SEND_PENDING	1	Packet queued pending an ARP reply.
IP_ERR_NOT_MINE	2	Packet was not of interest (upcall reply).

Table 5.24: embOS/IP error types

Chapter 6

TCP zero-copy interface

The TCP protocol can be used via socket functions or the TCP zero-copy interface which is described in this chapter.

6.1 TCP zero-copy

This section documents an optional extension to the Sockets layer, the TCP zero-copy API. The TCP zero-copy API is intended to assist the development of higher-performance embedded network applications by allowing the application direct access to the TCP/IP stack packet buffers. This feature can be used to avoid the overhead of having the stack copy data between application-owned buffers and stack-owned buffers in `send()` and `recv()`, but the application has to fit its data into, and accept its data from, the stack buffers.

The TCP zero-copy API is small because it is simply an extension to the existing Sockets API that provides an alternate mechanism for sending and receiving data on a socket. The Sockets API is used for all other operations on the socket.

6.1.1 Allocating, freeing and sending packet buffers

The two functions for allocating and freeing packet buffers are straightforward requests:

`IP_TCP_Alloc()` allocates a packet buffer from the pool of packet buffers on the stack and `IP_TCP_Free()` frees a packet buffer. Applications using the TCP zero-copy API are responsible for allocating packet buffers for use in sending data, as well as for freeing buffers that have been used to receive data and those that the application has allocated but decided not to use for sending data. As these packet buffers are a limited resource, it is important that applications free them promptly when they are no longer of use.

The functions for sending data, `IP_TCP_Send() and IP_TCP_SendAndFree()`, send a packet buffer of data using a socket. The TCP zero-copy interface supports two different approaches to send and free a packet. One approach is that the stack frees the packet independent from the success of sending the packet. Therefor, `IP_TCP_SendAndFree()` is called to send and free the packet. It frees the packet independent from the success of the send operation. The other approach is that `IP_TCP_Send()` is called. In this case it is the responsibility of the application to free the packet. Depending on the return value the application can decide if `IP_TCP_Free()` should be called to free the packet.

6.1.2 Callback function

Applications that use the TCP Zero-copy API for receiving data must include a callback function for acceptance of received packets, and must register the callback function with the socket using the `setsockopt()` sockets function with the `SO_CALLBACK` option name. The callback function, once registered, receives not only received data packets, but also connection events that result in socket errors.

6.2 Sending data with the TCP zero-copy API

To send data with the TCP zero-copy API, you should proceed as follow:

1. Allocating a packet buffer
2. Filling the allocated buffer
3. Sending the packet

The following section describes the procedure for allocating a packet buffer, sending data, and freeing the packet buffer step by step.

6.2.1 Allocating a packet buffer

The first step in using the TCP zero-copy API to send data is to allocate a packet buffer from the stack using the `IP_TCP_Alloc()` function. This function takes the maximum length of the data you intend to send in the buffer as argument and returns a pointer to an `IP_PACKET` structure.

```
IP_PACKET * pPacket;
U32         DataLen;             // Amount of data to send

DataLen = 512;                   // Should indicate amount of data to send
pPacket = IP_TCP_Alloc(DataLen);
if (pPacket == NULL) {
  // Error, could not allocate packet buffer
}
```

This limits how much data you can send in one call using the TCP zero-copy API, as the data sent in one call to `IP_TCP_Send()` must fit in a single packet buffer. The actual limit is determined by the big packet buffer size, less 68 bytes for protocol headers. If you try to request a larger buffer than this, `IP_TCP_Alloc()` returns NULL to indicate that it cannot allocate a sufficiently large buffer.

6.2.2 Filling the allocated buffer with data

Having allocated the packet buffer, you now fill it with the data to send. The function `IP_TCP_Alloc()` has initialized the returned `IP_PACKET` pPacket and so `pPacket->pData` points to where you can start depositing data.

6.2.3 Sending the packet

Finally, you send the packet by giving it back to the stack using the function `IP_TCP_Send()`.

```
e = IP_TCP_Send(socket, pPacket);
if (e < 0) {
  IP_TCP_Free(pPacket);
}
```

This function sends the packet over TCP, or returns an error. If its return value is less than zero, it has not accepted the packet and the application has to decide either to free the packet or to retain it for sending later. Use `IP_TCP_SendAndFree()` if the packet should be freed automatically in any case.

6.3 Receiving data with the TCP zero-copy API

To receive data with the TCP zero-copy API, you should proceed as follow:

1. Writing a callback function
2. Registering the callback function

6.3.1 Writing a callback function

Using the TCP zero-copy API for receiving data requires the application developer to write a callback function that the stack can use to inform the application of received data packets and other socket events. This function is expected to conform to the following prototype:

```
int rx_callback(long Socket, IP_PACKET * pPacket, int code);
```

The stack calls this function when it has received a data packet or other event to report for a socket. The parameter `Socket` identifies the socket. The parameter `pPacket` passes a pointer to the packet buffer (if there is a packet buffer). If pPacket is not NULL, it is a pointer to a packet buffer containing received data for the socket. `pPacket->pData` points to the start of the received data, and `pPacket->NumBytes` indicates the number of bytes of received data in this buffer.

The parameter code passes an error event (if there is an error to report). If code is not 0, it is a socket error indicating that an error or other event has occurred on the socket. Typical nonzero values are `ESHUTDOWN` and `ECONNRESET`. `ESHUTDOWN` defines that the connected peer has closed its end of the connection and sends no more data. `ECONNRESET` defines that the connected peer has abruptly closed its end of the connection and neither sends nor receives more data.

Returned values

The callback function may return one of the following values:

Symbolic	Numerical	Description
IP_OK	0	Data handled, packet can be freed.
IP_OK_KEEP_PACKET	1	Data will be handled by application later, the stack should NOT free the packet. This will be done by the application at a later time when the data has been handled and the packet is no longer needed.

Table 6.1: embOS/IP TCP zero-copy - Valid return values for the receive callback function

Note: The callback function is called from the stack and is expected to return promptly. Some of the places where the stack calls the callback function require that the data structures on the stack remain consistent through the callback, so the callback function must not call back into the stack except to call `IP_TCP_Free()`.

6.3.2 Registering the callback function

The application must also inform the stack of the callback function. `setsockopt()` function provides an additional socket option, `SO_CALLBACK`, which should be used for this purpose once the socket has been created. The following code fragment illustrates the use of this option to register a callback function named `RxUpcall()` on the socket `Socket`:

```
setsockopt(Socket, SOL_SOCKET, SO_CALLBACK, (void *)RxUpcall, 0);
```

The function `setsockopt()` is described in *setsockopt()* on page 132.

6.4 API functions

Function	Description
IP_TCP_Alloc()	Allocates a packet buffer.
IP_TCP_Free()	Frees a packet buffer.
IP_TCP_Send()	Sends a packet.
IP_TCP_SendAndFree()	Sends and frees a packet.

Table 6.2: embOS/IP TCP zero-copy API function overview

6.4.1 IP_TCP_Alloc()

Description

Allocates a packet buffer large enough to hold `datasize` bytes of TCP data, plus TCP, IP and MAC headers.

Prototype

```
IP_PACKET * IP_TCP_Alloc (int datasize);
```

Parameter

Parameter	Description
`datasize`	[IN] Length of the data which should be sent.

Table 6.3: IP_TCP_Alloc() parameter list

Return value

Success: Returns a pointer to the allocated buffer.
Error: NULL

Additional information

This function must be called to allocate a buffer for sending data via `IP_TCP_Send()`. It returns the allocated packet buffer with its `pPacket->pData` field set to where the application must deposit the data to be sent.

This `datasize` limits how much data that you can send in one call using the TCP zero-copy API, as the data sent in one call to `IP_TCP_Send()` must fit in a single packet buffer, with the TCP, IP, and lower-layer headers that the stack needs to add in order to send the packet.

The actual limit is determined by the big packet buffer size (normally 1536 bytes). Refer to *IP_AddBuffers()* on page 45 for more information about defining buffer sizes. If you try to request a larger buffer than this, `IP_TCP_Alloc()` returns NULL to indicate that it cannot allocate a sufficiently-large buffer.

Example

```
IP_PACKET * pPacket;
U32 DataLen;                // Amount of data to send

DataLen = 1024;             // Should indicate amount of data to send
pPacket = IP_TCP_Alloc(DataLen);
if (pPacket == NULL) {
   // Error, could not allocate packet buffer
}
```


6.4.2 IP_TCP_Free()

Description

Frees a packet buffer allocated by `IP_TCP_Alloc()`.

Prototype

```
void IP_TCP_Free ( IP_PACKET * pPacket );
```

Parameter

Parameter	Description
`pPacket`	[IN] Pointer to the `IP_Packet` structure.

Table 6.4: IP_TCP_Free() parameter list

6.4.3 IP_TCP_Send()

Description

Sends a packet buffer on a socket.

Prototype

```
int IP_TCP_Send ( U32         s,
                  IP_PACKET * pPacket );
```

Parameter

Parameter	Description
s	[IN] Socket descriptor.
pPacket	[IN] Pointer to a packet buffer.

Table 6.5: IP_TCP_Send() parameter list

Return value

0 The packet was sent successfully.
<0 The packet was not accepted by the stack. The application must re-send the packet using a call to `IP_TCP_Send()`, or free the packet using `IP_TCP_Free()`.
>0 The packet has been accepted and queued on the socket but has not yet been transmitted.

Additional information

Applications using the TCP zero-copy API are responsible for allocating packet buffers for use in sending data, as well as for freeing buffers that have been used to receive data and those that the application has allocated but decided not to use for sending data. As these packet buffers are a limited resource, it is important that applications free them promptly when they are no longer of use.

Packets have to be freed after processing. The TCP zero-copy interface supports two different approaches to free a packet. One approach is that the stack frees the packet independent from the success of sending the packet. Therefor, `IP_TCP_SendAndFree()` is called to send the packet and free the packet. It frees the packet independent from the success of the send operation. The other approach is that `IP_TCP_Send()` is called. In this case it is the responsibility application programmer to free the packet. Depending on the return value the application programmer can decide if `IP_TCP_Free()` should be called to free the packet.

6.4.4 IP_TCP_SendAndFree()

Description

Sends a packet buffer on a socket.

Prototype

```
int IP_TCP_SendAndFree ( U32         s,
                         IP_PACKET * pPacket );
```

Parameter

Parameter	Description
s	[IN] Socket descriptor.
pPacket	[IN] Pointer to the IP_Packet structure.

Table 6.6: IP_TCP_Send() parameter list

Return value

0 The packet was sent successfully.
<0 The packet was not accepted by the stack.
>0 The packet has been accepted and queued on the socket but has not yet been transmitted.

Additional information

Applications using the TCP zero-copy API are responsible for allocating packet buffers for use in sending data, as well as for freeing buffers that have been used to receive data and those that the application has allocated but decided not to use for sending data. As these packet buffers are a limited resource, it is important that applications free them promptly when they are no longer of use.

IP_TCP_SendAndFree() frees packet pPacket after processing. It frees the packet independent from the success of the send operation.

Chapter 7

UDP zero-copy interface

The UDP transfer protocol can be used via socket functions or the zero-copy interface which is described in this chapter.

7.1 UDP zero-copy

The UDP zero-copy API functions are provided for systems that do not need the overhead of sockets. These routines impose a lower demand on CPU and system memory requirements than sockets. However, they do not offer the portability of sockets.

UDP zero-copy API functions are intended to assist the development of higher-performance embedded network applications by allowing the application direct access to the TCP/IP stack packet buffers. This feature can be used to avoid the overhead of having the stack copy data between application-owned buffers and stack-owned buffers in `sendto()` and `recvfrom()`, but the application has to fit its data into, and accept its data from, the stack buffers. Refer to *embOS/IP UDP discover (OS_IP_UDPDiscover.c / OS_IP_UDPDiscoverZeroCopy.c)* on page 39 for detailed dinformation about the UDP zero-copy example application.

7.1.1 Allocating, freeing and sending packet buffers

The two functions for allocating and freeing packet buffers are straightforward requests:

`IP_UDP_Alloc()` allocates a packet buffer from the pool of packet buffers on the stack and `IP_UDP_Free()` frees a packet buffer. Applications using the UDP zero-copy API are responsible for allocating packet buffers for use in sending data, as well as for freeing buffers that have been used to receive data and those that the application has allocated but decided not to use for sending data. As these packet buffers are a limited resource, it is important that applications free them promptly when they are no longer of use.

The functions for sending data, `IP_UDP_Send() and IP_UDP_SendAndFree()`, send a packet buffer of data using a port. The UDP zero-copy interface supports two different approaches to send and free a packet. One approach is that the stack frees the packet independent from the success of sending the packet. Therefor, `IP_UDP_SendAndFree()` is called to send and free the packet. It frees the packet independent from the success of the send operation. The other approach is that `IP_UDP_Send()` is called. In this case it is the responsibility of the application to free the packet. Depending on the return value the application can decide if `IP_UDP_Free()` should be called to free the packet.

7.1.2 Callback function

Applications that use the UDP zero-copy API for receiving data must include a callback function for acceptance of received packets, and must register the callback function with a port using the `IP_UDP_Open()` function. The callback function, once registered, receives all matching data packets.

7.2 Sending data with the UDP zero-copy API

To send data with the UDP zero-copy API, you should proceed as follow:

1. Allocating a packet buffer
2. Filling the allocated buffer
3. Sending the packet

The following section describes the procedure for allocating a packet buffer, sending data, and freeing the packet buffer step by step.

7.2.1 Allocating a packet buffer

The first step in using the UDP zero-copy API to send data is to allocate a packet buffer from the stack using the `IP_UDP_Alloc()` function. This function takes the maximum length of the data you intend to send in the buffer as argument and returns a pointer to an `IP_PACKET` structure.

```
IP_PACKET * pPacket;
U32         DataLen;             // Amount of data to send

DataLen = 512;                   // Should indicate amount of data to send
pPacket = IP_UDP_Alloc(DataLen);
if (pPacket == NULL) {
  // Error, could not allocate packet buffer
}
```

This limits how much data you can send in one call using the UDP zero-copy API, as the data sent in one call to `IP_UDP_Send()` must fit in a single packet buffer. The actual limit is determined by the big packet buffer size, less typically 42 bytes for protocol headers (14 bytes for Ethernet header, 20 bytes IP header, 8 bytes UDP header). If you try to request a larger buffer than this, `IP_UDP_Alloc()` returns NULL to indicate that it cannot allocate a sufficiently large buffer.

7.2.2 Filling the allocated buffer with data

Having allocated the packet buffer, you now fill it with the data to send. The function `IP_UDP_Alloc()` has initialized the returned `IP_PACKET pPacket` and so `pPacket->pData` points to where you can start depositing data.

7.2.3 Sending the packet

Finally, you send the packet by giving it back to the stack using the function `IP_UDP_Send()`.

```
#define SRC_PORT  50020
#define DEST_PORT 50020
#define DEST_ADDR 0xC0A80101

e = IP_UDP_Send(0, DEST_ADDR, SRC_PORT, DEST_PORT, pPacket);
if (e < 0) {
  IP_UDP_Free(pPacket);
}
```

This function sends the packet over UDP, or returns an error. If its return value is less than zero, it has not accepted the packet and the application has to decide either to free the packet or to retain it for sending later. Use `IP_UDP_SendAndFree()` if the packet should be freed automatically in any case.

7.3 Receiving data with the UDP zero-copy API

To receive data with the UDP zero-copy API, you should proceed as follow:

1. Writing a callback function
2. Registering the callback function

7.3.1 Writing a callback function

Using the UDP zero-copy API for receiving data requires the application developer to write a callback function that the stack can use to inform the application of received data packets. This function is expected to conform to the following prototype:

```
int rx_callback(IP_PACKET * pPacket, void * pContext)
```

The stack calls this function when it has received a data packet for a port. The parameter `pPacket` points to the packet buffer. The packet buffer contains the received data for the socket. `pPacket->pData` points to the start of the received data, and `pPacket->NumBytes` indicates the number of bytes of received data in this buffer.

Returned values

The callback function may return one of the following values:

Symbolic	Numerical	Description
IP_OK	0	Data handled. embOS/IP will free the packet.
IP_OK_KEEP_PACKET	1	Data will be handled by application later, the stack should NOT free the packet. This will be done by the application at a later time when the data has been handled and the packet is no longer needed.

Table 7.1: embOS/IP UDP zero-copy - Valid return values for the receive callback function

Note: The callback function is called from the stack and is expected to return promptly. Some of the places where the stack calls the callback function require that the data structures on the stack remain consistent through the callback, so the callback function must not call back into the stack except to call `IP_UDP_Free()`.

7.3.2 Registering the callback function

The application must also inform the stack of the callback function. This is done by calling the `IP_UDP_Open()` function. The following code fragment illustrates the use of this option to register a callback function named `RxUpcall()` on the port `50020`:

```
#define SRC_PORT  50020
#define DEST_PORT 50020

IP_UDP_Open(0L /* any foreign host */, SRC_PORT, DEST_PORT, RxUpCall, 0L /* any tag
*/);
```

The function `IP_UDP_Open()` is described in *IP_UDP_Open()* on page 166.

7.4 API functions

Function	Description
IP_UDP_Alloc()	Returns a pointer to a packet buffer big enough for the specified sizes.
IP_UDP_Close()	Closes a UDP connection handle.
IP_UDP_FindFreePort()	Returns a free local port number.
IP_UDP_Free()	Frees the buffer which was used for a packet.
IP_UDP_GetDataPtr()	Returns pointer to data contained in the received UDP packet.
IP_UDP_GetFPort()	Extracts foreign port information from a UDP packet.
IP_UDP_GetLPort()	Extracts local port information from a UDP packet.
IP_UDP_GetSrcAddr()	Retrieves the IP address of the sender of the given UDP packet.
IP_UDP_Open()	Creates a UDP connection handle.
IP_UDP_Send()	Sends an UDP packet to a specified host.
IP_UDP_SendAndFree()	Sends an UDP packet to a specified host and frees the packet.

Table 7.2: embOS/IP UDP zero-copy API function overview

7.4.1 IP_UDP_Alloc()

Description

Returns a pointer to a packet buffer big enough for the specified sizes.

Prototype

```
IP_PACKET * IP_UDP_Alloc( int NumBytes );
```

Parameter

Parameter	Description
NumBytes	[IN] Length of the data which should be sent.

Table 7.3: IP_UDP_Alloc() parameter list

Return value

Success: Returns a pointer to the allocated buffer.
Error: NULL

Additional information

Applications using the UDP zero-copy API are responsible for allocating packet buffers for use in sending data, as well as for freeing buffers that have been used to receive data and those that the application has allocated but decided not to use for sending data. As these packet buffers are a limited resource, it is important that applications free them promptly when they are no longer of use.

The UDP zero-copy interface supports two different approaches to free a packet. One approach is that the stack frees the packet independent from the success of sending the packet. Therefor, `IP_UDP_SendAndFree()` is called to send the packet and free the packet. It frees the packet independent from the success of the send operation. The other approach is that `IP_UDP_Send()` is called. In this case it is the responsibility application programmer to free the packet. Depending on the return value the application programmer can decide if `IP_UDP_Free()` should be called to free the packet.

7.4.2 IP_UDP_Close()

Description

Closes a UDP connection handle and removes the connection from demux table list of connections and deallocates it.

Prototype

```
void IP_UDP_Close( IP_UDP_CONN Con );
```

Parameter

Parameter	Description
Con	[IN] UDP connection handle.

Table 7.4: IP_UDP_Close() parameter list

7.4.3 IP_UDP_FindFreePort()

Description

Obtains a random port number. that is suitable for use as the `lport` parameter in a call to `IP_UDP_Open()`.

Prototype

```
U16 IP_UDP_FindFreePort( void );
```

Return value

A usable port number in local endianess.

Additional information

The returned port number is suitable for use as the `lport` parameter in a call to `IP_UDP_Open()`. Refer to *IP_UDP_Open()* on page 166 for more information. `IP_UDP_FindFreePort()` avoids picking port numbers in the reserved range 0-1024, or in the range 1025-1199, which may be used for server applications.

7.4.4 IP_UDP_Free()

Description

Frees the buffer which was used for a packet.

Prototype

```
void IP_UDP_Free( IP_PACKET * pPacket );
```

Parameter

Parameter	Description
pPacket	[IN] Pointer to a packet structure.

Table 7.5: IP_UDP_Free() parameter list

7.4.5 IP_UDP_GetDataPtr()

Description

Returns pointer to data contained in the received UDP packet.

Prototype

```
void * IP_UDP_GetDataPtr( const IP_PACKET * pPacket );
```

Parameter

Parameter	Description
pPacket	[IN] Pointer to a packet structure.

Table 7.6: IP_UDP_GetDataPtr() parameter list

7.4.6 IP_UDP_GetFPort()

Description

Extracts foreign port information from a UDP packet.

Prototype

```
U16 IP_UDP_GetFPort ( const IP_PACKET * pPacket );
```

Parameter

Parameter	Description
pPacket	[IN] Pointer to a packet structure.

Table 7.7: IP_UDP_GetFPort() parameter list

7.4.7 IP_UDP_GetLPort()

Description

Extracts local port information from a UDP packet.

Prototype

```
U16 IP_UDP_GetLPort ( const IP_PACKET * pPacket );
```

Parameter

Parameter	Description
pPacket	[IN] Pointer to a packet structure.

Table 7.8: IP_UDP_GetLPort() parameter list

7.4.8 IP_UDP_GetSrcAddr()

Description

Returns pointer to data contained in the received UDP packet.

Prototype

```
void IP_UDP_GetSrcAddr( const IP_PACKET * pPacket,
                              void      * pSrcAddr,
                              int         AddrLen );
```

Parameter

Parameter	Description
pPacket	[IN] Pointer to a packet structure.
pSrcAddr	[IN] Pointer to a buffer to store the source address.
AddrLen	[IN] Size of the buffer used to store the source address.

Table 7.9: IP_UDP_GetSrcAddr() parameter list

7.4.9 IP_UDP_Open()

Description

Creates a UDP connection handle to receive, and pass upwards, UDP packets that match the parameters passed.

Prototype

```
IP_UDP_CONN IP_UDP_Open( IP_ADDR         IPAddr,
                         U16             fport,
                         U16             lport,
                         int(*routine)   (IP_PACKET *, void * pContext),
                         void *          pContext );
```

Parameter

Parameter	Description
IPAddr	[IN] IP address.
fport	[IN] Foreign port.
lport	[IN] Local port.
(*routine)	[IN] Callback function which is called when a UDP packet is received.
pContext	[IN/OUT] Application defined context pointer.

Table 7.10: IP_UDP_Open() parameter list

Return value

Success: Returns a pointer to the UDP connection handle.
Error: NULL

Additional information

The parameters IPAddr, fport, and lport, can be set to 0 as a wild card, which enables the reception of broadcast datagrams. The callback handler function is called with a pointer to a received datagram and a copy of the data pointer which is passed to IP_UDP_Open(). This can be any data the programmer requires, such as a pointer to another function, or a control structure to aid in demultiplexing the received UDP packet.

The returned handle is used as parameter for IP_UDP_Close() only. If IP_UDP_Close() is not called, there is no need to safe the return value.

7.4.10 IP_UDP_Send()

Description

Send an UDP packet to a specified host.

Prototype

```
int IP_UDP_Send( int          IFace,
                 IP_ADDR      FHost,
                 U16          fport,
                 U16          lport,
                 IP_PACKET * pPacket );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based index of available interfaces.
IPAddr	[IN] IP address of the target host in network endianess.
fport	[IN] Foreign port.
lport	[IN] Local port.
pPacket	[IN] Data which should be sent to the target host.

Table 7.11: IP_UDP_Send() parameter list

Return value

On success: 0
On error: Non-zero error code

Additional information

The packet pPacket has to be allocated by calling IP_UDP_Alloc(). Refer to *IP_UDP_Alloc()* on page 158 for detailed information.

If you expect to get any response to this packet you should have opened a UDP connection prior to calling IP_UDP_Send(). Refer to *IP_UDP_Open()* on page 166 for more information about creating an UDP connection.

IP_UDP_Send() does not free the packet after sending. It is the responsibility of the application programmer to free the packet. Depending on the return value the application programmer can decide if IP_UDP_Free() should be called to free the packet.

7.4.11 IP_UDP_SendAndFree()

Description

Send an UDP packet to a specified host and frees the packet.

Prototype

```
int IP_UDP_SendAndFree( int         IFace,
                        IP_ADDR     FHost,
                        U16         fport,
                        U16         lport,
                        IP_PACKET * pPacket );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based index of available interfaces.
IPAddr	[IN] IP address of the target host in network endianess.
fport	[IN] Foreign port.
lport	[IN] Local port.
pPacket	[IN] Data which should be sent to the target host.

Table 7.12: IP_UDP_SendAndFree() parameter list

Return value

On success: 0
On error: Non-zero error code

Additional information

The packet pPacket has to be allocated by calling IP_UDP_Alloc(). Refer to IP_UDP_Alloc() on page 72 for detailed information.

If you expect to get any response to this packet you should have opened a UDP connection prior to calling this. Refer to *IP_UDP_Open()* on page 166 for more information about creating an UDP connection.

Packets are freed by calling IP_UDP_SendAndFree(). Therefor, no call of IP_UDP_Free() is required.

Chapter 8

DHCP client

This chapter explains the usage of the Dynamic Host Control Protocol (DHCP) with embOS/IP. All API functions are described in this chapter.

8.1 DHCP backgrounds

DHCP stands for Dynamic Host Configuration Protocol. It is designed to ease configuration management of large networks by allowing the network administrator to collect all the IP hosts "soft" configuration information into a single computer. This includes IP address, name, gateway, and default servers. Refer to *[RFC 2131] - DHCP - Dynamic Host Configuration Protocol* for detailed information about all settings which can be assigned with DHCP.

DHCP is a "client/server" protocol, meaning that machine with the DHCP database "serves" requests from DHCP clients. The clients typically initiate the transaction by requesting an IP address and perhaps other information from the server. The server looks up the client in its database, usually by the client's media address, and assigns the requested fields. Clients do not always need to be in the server's database. If an unknown client submits a request, the server may optionally assign the client a free IP address from a "pool" of free addresses kept for this purpose. The server may also assign the client default information of the local network, such as the default gateway, the DNS server, and routing information.

When the IP addresses is assigned, it is "leased" to the client for a finite amount of time. The DHCP client needs to keep track of this lease time, and obtain a lease extension from the server before the lease time runs out. Once the lease has elapsed, the client should not send any more IP packets (except DHCP requests) until he get another address. This approach allows computers (such as laptops or factory floor monitors) which will not be permanently attached to the network to share IP addresses and not hog them when they are not using the net.

DHCP is just a superset of the Bootstrap Protocol (BOOTP). The main differences between the two are the lease concept, which was created for DHCP, and the ability to assigned addresses from a pool. Refer to *[RFC 951] - Bootstrap Protocol* for detailed information about the Bootstrap Protocol.

8.2 API functions

Function	Description
IP_DHCPC_Activate()	Activates the DHCP client.
IP_DHCPC_GetState()	Returns the state of the DHCP client.
IP_DHCPC_Halt()	Stops all DHCP client activity.
IP_DHCPC_SetCallback()	Sets a callback for an interface.

Table 8.1: embOS/IP DHCP client interface function overview

8.2.1 IP_DHCPC_Activate()

Description

Activates the DHCP client.

Prototype

```
void IP_DHCPC_Activate ( int          IFIndex,
                         const char * sHost,
                         const char * sDomain,
                         const char * sVendor );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based index number specifying the interface which should request configuration information from a DHCP server.
sHost	[IN] Pointer to host name to use in negotiation. Can be NULL.
sDomain	[IN] Pointer to domain name to use in negotiation. Can be NULL.
sVendor	[IN] Pointer to vendor to use in negotiation. Can be NULL.

Table 8.2: IP_DHCPC_Activate() parameter list

Additional information

This function is typically called from within `IP_X_Config()`. This function initializes the DHCP client. It attempts to open a UDP connection to listen for incoming replies and begins the process of configuring a network interface using DHCP. The process may take several seconds, and the DHCP client will keep retrying if the service does not respond.

The parameters `sHost`, `sDomain`, `sVendor` are optional (can be NULL). If not NULL, must point to a memory area which remains valid after the call since the string is not copied.

Example

```
// Correct function call
IP_DHCPC_Activate(0, "Target", NULL, NULL);
// Illegal function call
char ac;
sprintf(ac, "Target%d, Index);
IP_DHCPC_Activate(0, ac, NULL, NULL);
// Correct function call
static char ac;
sprintf(ac, "Target%d, Index);
IP_DHCPC_Activate(0, ac, NULL, NULL);
```

If you start the DHCP client with activated logging the output on the terminal I/O should be similar to the listing below:

```
DHCP: Sending discover!
DHCP: Received packet from 192.168.1.1
DHCP: Packet type is OFFER.
DHCP: Renewal time: 2160 min.
DHCP: Rebinding time: 3780 min.
DHCP: Lease time: 4320 min.
DHCP: Host name received.
DHCP: Sending Request.
DHCP: Received packet from 192.168.1.1
DHCP: Packet type is ACK.
DHCP: Renewal time: 2160 min.
DHCP: Rebinding time: 3780 min.
DHCP: Lease time: 4320 min.
DHCP: Host name received.
DHCP: IFace 0: IP: 192.168.199.20, Mask: 255.255.0.0, GW: 192.168.1.1.
```


8.2.2 IP_DHCPC_GetState()

Description

Returns the state of the DHCP client.

Prototype

```
int IP_DHCPC_GetState( int IFIndex );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based index number specifying the interface for which the state should be requested.

Table 8.3: IP_DHCPC_GetState() parameter list

Return value

0 DHCP client not used.
>0 DHCP client in use.

8.2.3 IP_DHCPC_Halt()

Description

Stops all DHCP activity on a network interface.

Prototype

```
void IP_DHCPC_Halt( int IFIndex );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based index number specifying the interface which should be halted.

Table 8.4: IP_DHCPC_Halt() parameter list

8.2.4 IP_DHCPC_SetCallback()

Description

This function allows the caller to set a callback for an interface.

Prototype

```
void IP_DHCPC_SetCallback( int IFIndex, int (*routine)(int,int) );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based index number of available network interfaces.
(*routine)	[IN] Callback functions which should be called with every status change.

Table 8.5: IP_DHCPC_SetCallback() parameter list

Additional information

The callback is called with every status change. This mechanism is provided so that the caller can do some processing when the interface is up (like doing initialization or blinking LEDs, etc.). Refer to *[RFC 2331] DHCP - Dynamic Host Configuration Protocol* for detailed information about DHCP states.

Chapter 9

AutoIP

All functions which are required to add AutoIP to your application are described in this chapter.

9.1 embOS/IP AutoIP backgrounds

The embOS/IP AutoIP module adds the dynamic configuration of IPv4 Link-Local addresses to embOS/IP. This functionality is better known as AutoIP. Therefore, this term will be used in this document.

The AutoIP implementation covers the relevant parts of the following RFCs:

RFC#	Description
[RFC 3972]	Dynamic Configuration of IPv4 Link-Local Addresses. Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc3972.txt*

In general AutoIP is a method to negotiate a IPv4 address in a network without the utilization of a server such as a DHCP server. AutoIP will try to use IPv4 addresses out of a reserved pool from the addresses 169.254.1.0 to 169.254.254.255 to find a free IP that is not used by any other network participant at this time.

To achieve this goal AutoIP sends ARP probes into the network to ask if the addr. to be used is already in use. This is determined by an ARP reply for the requested address. Upon an address conflict AutoIP will generate a new address to use and will retry to use it by sending ARP probes again.

9.2 API functions

Function	Description
IP_AutoIP_Activate()	Activates AutoIP.
IP_AutoIP_Halt()	Stops all AutoIP activity.
IP_AutoIP_SetUserCallback()	Sets a callback to get a notification about each status change.
IP_AutoIP_SetStartIP()	Sets the IP address which will be used for the first configuration try.

Table 9.1: embOS/IP AutoIP interface function overview

9.2.1 IP_AutoIP_Activate()

Description

Activates AutoIP for the specified interface.

Prototype

```
void IP_AutoIP_Activate ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.

Table 9.2: IP_AutoIP_Activate() parameter list

Additional information

Activating the dynamic configuration of IPv4 Link-Local addresses means that an additional timer will be added to the stack. This timer will be called every second to check the status of the address configuration. With the AutoIP activation an IP address for the dynamic configuration will be created. The IPv4 prefix 169.254/16 is registered with the IANA for this purpose. This means that embOS/IP will generate an IP address similar to 169.254.xxx.xxx. The subnet mask of is always 255.255.0.0.

In embOS/IP debug builds terminal I/O output can be enabled. AutoIP outputs status information in the terminal I/O window if the stack is configured to so (IP_MTYPE_AUTOIP added to the log filter mask). Please refer to *IP_SetLogFilter()* on page 439 and *IP_AddLogFilter()* on page 437 for further information about the enabling terminal I/O. If terminal I/O is enabled the output of a the program start should be similar to the following lines:

```
0:000 MainTask - INIT: Init started. Version 2.00.06
0:000 MainTask - DRIVER: Found PHY with Id 0x2000 at addr 0x1
0:000 MainTask - INIT: Link is down
0:000 MainTask - INIT: Init completed
0:000 IP_Task - INIT: IP_Task started
0:000 IP_RxTask - INIT: IP_RxTask started
3:000 IP_Task - LINK: Link state changed: Full duplex, 100 MHz
9:000 IP_Task - AutoIP: 169.254.240.240 checked, no conflicts
9:000 IP_Task - AutoIP: IFaceId 0: Using IP: 169.254.240.240.
```


9.2.2 IP_AutoIP_Halt()

Description

Stops AutoIP activity for the passed interface.

Prototype

```
void IP_AutoIP_Halt ( unsigned IFaceId
                      U8       KeepIP );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
KeepIP	[IN] Flag to indicate if the used IP address should be stored for the next start of AutoIP. 0 means do not keep the IP, 1 means keep the IP address for the next AutoIP start.

Table 9.3: IP_AutoIP_Halt() parameter list

Return value

0 : Ok. AutoIP stopped. IP address cleared.
IP : Ok. AutoIP stopped. The IP address (for example, 0xA9FExxxx) has been kept.
-1 : Error. Illegal interface number.

Additional information

The function stops the AutoIP module. The IP address which was used during AutoIP was activated, can be kept to speed up the configuration process after reactivating AutoIP. If the IP address will not be kept, AutoIP creates a new IP address after the reactivation.

9.2.3 IP_AutoIP_SetUserCallback()

Description

Sets a callback function. It will be called with every status change.

Prototype

```
void IP_AutoIP_SetUserCallback( unsigned                     IFaceId,
                                IP_AUTOIP_INFORM_USER_FUNC * pfInformUser );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
pfInformUser	[IN] Pointer to a user function of type IP_AUTOIP_INFORM_USER_FUNC which is called when a status change occurs.

Table 9.4: IP_AutoIP_SetCallback() parameter list

Additional Information

The possibility to set a callback function is provided so that the caller can do some processing when the interface is up (like doing initializations or blinking LEDs, etc.).

IP_AUTOIP_INFORM_USER_FUNC is defined as follows:

```
typedef void (IP_AUTOIP_INFORM_USER_FUNC)(U32 IFaceId, U32 Status);
```


9.2.4 IP_AutoIP_SetStartIP()

Description

Sets the IP address which will be used for the first configuration try.

Prototype

```
void IP_AutoIP_SetStartIP( unsigned IFaceId,
                           U32      IPAddr );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
IPAddr	[IN] 4-byte IPv4 address.

Table 9.5: IP_AutoIP_SetCallback() parameter list

Additional information

A call of this function is normally not required, but in some cases it can be useful to set the IP address which should be used as starting point of the AutoIP functionality.

9.3 AutoIP resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the AutoIP module presented in the tables below have been measured on an ARM7 and a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

9.3.1 ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP AutoIP module	approximately 1.1Kbyte

Table 9.6: AutoIP ROM usage ARM7

9.3.2 ROM usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

Addon	ROM
embOS/IP AutoIP module	approximately 1.0Kbyte

Table 9.7: AutoIP ROM usage Cortex-M3

9.3.3 RAM usage

Addon	RAM
embOS/IP AutoIP module	approximately 0.7Kbyte

Table 9.8: AutoIP RAM usage

Chapter 10

Address Collision Detection

All functions which are required to add Address Collision Detection (ACD) to your application are described in this chapter.

10.1 embOS/IP ACD backgrounds

The embOS/IP ACD module allows the user specific configuration of the behavior if an IPv4 address collision is detected. This means that more than one host in the network is using the same IPv4 address at the same time. This is discovered sending ARP discover packets to find hosts with the same addresses in the network.

10.2 API functions

Function	Description
IP_ACD_Activate()	Activates ACD.
IP_ACD_Config()	Configures parameter for ACD.

Table 10.1: embOS/IP ACD interface function overview

10.2.1 IP_ACD_Activate()

Description

Activates ACD for the specified interface.

Prototype

```
int IP_ACD_Activate ( unsigned IFace );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based interface index.

Table 10.2: IP_ACD_Activate() parameter list

Return value

0 ACD activated and free IP found (does not mean the initial IP was good)
1 No IP address set when ACD was activated

Additional information

Activating the address conflict detection module means that a a hook into the ARP module of the stack will be activated that allows the user to take action if an IPv4 address conflict on the network has been discovered.

When the ACD module is started it will check if the currently used IP address is in conflict with any other host on the network by sending ARP probes to find hosts with the same IPv4 address.

To allow the user to take action on those conflicts it is necessary to use *IP_ACD_Config()* on page 189 before activating ACD.

In embOS/IP debug builds terminal I/O output can be enabled. ACD outputs status information in the terminal I/O window if the stack is configured to so (IP_MTYPE_ACD added to the log filter mask). Please refer to *IP_SetLogFilter()* on page 439 and *IP_AddLogFilter()* on page 437 for further information about the enabling terminal I/O.

10.2.2 IP_ACD_Config()

Description

Configures ACD behavior for startup and in case of conflicts.

Prototype

```
void IP_ACD_Config ( unsigned          IFace
                     unsigned          ProbeNum
                     unsigned          DefendInterval
                     const ADC_FUNC *  pACDContext );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based interface index.
ProbeNum	[IN] Number of ARP probes to send upon activating ACD before declaring the actual used IP address to be free to be used.
DefendInterval	[IN] Interval in which the currently active IP address is being known as defended after taking action.
pACDContext	[IN] Pointer to a structure of type ACD_FUNC.

Table 10.3: IP_ACD_Config() parameter list

10.3 ACD data structures

10.3.1 Structure ACD_FUNC

Description

Used to store function pointers to the user defined callbacks to take several actions upon detecting an IP address conflict.

Prototype

```
typedef struct ACD_FUNC {
  U32 (*pfRenewIPAddr);
  int (*pfDefend);
  int (*pfRestart);
} ACD_FUNC;
```

Member	Description
pfRenewIPAddr	Function pointer to a user defined routine that is used to generate a new IPv4 address if there is a collision detected during ACD activation.
pfDefend	Function pointer to a user defined routine that is used to let the user implement his own defend strategy. Can be NULL.
pfRestart	Function pointer to a user defined routine that should reconfigure the IP address used by the stack and optionally re-activates ACD.

Table 10.4: Structure ACD_FUNC member list

10.4 ACD resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the AutoIP module presented in the tables below have been measured on an ARM7 and a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

10.4.1 ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP ACD module	approximately 0.4Kbyte

Table 10.5: ACD ROM usage ARM7

10.4.2 ROM usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

Addon	ROM
embOS/IP ACD module	approximately 0.4Kbyte

Table 10.6: ACD ROM usage Cortex-M3

10.4.3 RAM usage

Addon	RAM
embOS/IP ACD module	approximately 50Bytes

Table 10.7: ACD RAM usage

Chapter 11

UPnP (Add-on)

The embOS/IP implementation of UPnP which stand for Universal Plug and Play is an optional extension to embOS/IP. It allows making your target easily discoverable and advertising services available on your target throughout your network.

11.1 embOS/IP UPnP

The embOS/IP UPnP implementation is an optional extension which can be seamlessly integrated into your TCP/IP application. It combines the possibility to implemented UPnP services in a most flexible way by allowing to specify content to be sent upon UPnP requests completely generated by the application with a small memory footprint.

The UPnP module implements the relevant parts of the UPnP documentation released by the UPnP Forum.

Document	Download
UPnP Device Architecture 1.0	Direct download: *http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf*

The following table shows the contents of the embOS/IP root directory:

Directory	Content
`Application`	Contains the example application to run the UPnP implementation with embOS/IP and embOS/IP Web server add-on.
`IP`	Contains the UPnP source file, `IP_UPnP.c` .

Supplied directory structure of embOS/IP UPnP package

11.2 Feature list

- Low memory footprint.
- Advertising your services on the network
- Easy to implement

11.3 Requirements

TCP/IP stack

The embOS/IP UPnP implementation requires the embOS/IP TCP/IP stack and is designed to be used with the embOS/IP Web server add-on.

11.4 UPnP backgrounds

UPnP is designed to provide services throughout a network without interaction of the user. It is designed to use standardised protocols such as IP, TCP, UDP, Multicast, HTTP and XML for communication and to dirstribute services provided by a device.

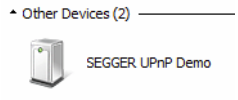
UPnP can be used to advertise services provided by a device across the network such as where to find the web interface for the device advertising. Newer operating systems support UPnP from scratch and will show UPnP devices available across a network and may provide easy access to a device by simply selecting the discovered UPnP device.

A typical usage would be to advertise media accessible on a media storage on the network and opening a file browser window to the resource upon opening the UPnP entry discovered.

11.4.1 Using UPnP to advertise your service in the network

The default UPnP XML file advertised is `upnp.xml`. A solution providing UPnP content has to serve a file called `upnp.xml` containing valid UPnP descriptors via a web server. The sample `OS_IP_WebserverUPnP.c` provides a sample configuration for advertising a web server page that will open if the UPnP client clicks on the discovered UPnP device.

A discovered UPnP device will typically be shown in the network neighborhood of your operating system. A discovered device found by a Windows OS is shown in the picture below:



The example below shows the most important excerpts from the OS_IP_WebserverUPnP.c sample that are necessary to setup a UPnP device in your network.

Example

The sample provides some easy to use defines to adopt the identification strings used by the UPnP device to advertise itself to be changed to your needs.

```
/* Excerpt from OS_IP_WebserverUPnP.c */
//
// UPnP
//
#define UPNP_FRIENDLY_NAME      "SEGGER UPnP Demo"
#define UPNP_MANUFACTURER       "SEGGER Microcontroller GmbH and Co. KG"  // '&' is
not allowed
#define UPNP_MANUFACTURER_URL   "http://www.segger.com"
#define UPNP_MODEL_DESC         "SEGGER Web server with UPnP"
#define UPNP_MODEL_NAME         "SEGGER UPnP Demo"
#define UPNP_MODEL_URL          "http://www.segger.com/embos-ip-webserver.html"
```

The sample uses VFile hooks as described in *IP_WEBS_AddVFileHook()* on page 306 to provide dynamically serving the necessary XML files for UPnP without the need for a real file system or further processing through the web server.

```
/* Excerpt from OS_IP_WebserverUPnP.c */
/*********************************************************************
*
*       Types
*
**********************************************************************
*/

typedef struct {
  const char     * sFileName;
  const char     * pData;
```



```
        unsigned   NumBytes;
} VFILE_LIST;

/* Excerpt from OS_IP_WebserverUPnP.c */
/*********************************************************************
*
*       Static const
*
**********************************************************************
*/

//
// UPnP, virtual files
//
static const char _acFile_dummy_xml[] =
  "<?xml version=\"1.0\" encoding=\"utf-8\"?>\r\n"
  "<scpd xmlns=\"urn:schemas-upnp-org:service-1-0\">\r\n"
    "<specVersion>\r\n"
      "<major>1</major>\r\n"
      "<minor>0</minor>\r\n"
    "</specVersion>\r\n"
    "<serviceStateTable>\r\n"
      "<stateVariable>\r\n"
        "<name>Dummy</name>\r\n"
        "<dataType>i1</dataType>\r\n"
      "</stateVariable>\r\n"
    "</serviceStateTable>\r\n"
  "</scpd>";

//
// UPnP, virtual files list
//
static const VFILE_LIST _VFileList[] = {
  "/dummy.xml", _acFile_dummy_xml, sizeof(_acFile_dummy_xml) - 1,  // Do not count in
the null terminator of the string
  NULL        , NULL             , NULL
};

/* Excerpt from OS_IP_WebserverUPnP.c */
//
// UPnP webserver VFile hook
//
static WEBS_VFILE_HOOK _UPnP_VFileHook;
```

Several helper functions are provided in the sample to easily generate a valid XML file for advertising a service using UPnP.

```
/* Excerpt from OS_IP_WebserverUPnP.c */
//
// UPnP
//
#define UPNP_FRIENDLY_NAME     "SEGGER UPnP Demo"
#define UPNP_MANUFACTURER       "SEGGER Microcontroller GmbH and Co. KG"   // '&' is
not allowed
#define UPNP_MANUFACTURER_URL  "http://www.segger.com"
#define UPNP_MODEL_DESC        "SEGGER Web server with UPnP"
#define UPNP_MODEL_NAME        "SEGGER UPnP Demo"
#define UPNP_MODEL_URL         "http://www.segger.com/embos-ip-webserver.html"

/* Excerpt from OS_IP_WebserverUPnP.c */
/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
```



```
*
*       _UPnP_GetURLBase
*
* Function description
*   This function copies the information needed for the URLBase parameter
*   into the given buffer and returns a pointer to the start of the buffer
*   for easy readable code.
*
* Parameters
*   pBuffer        - Pointer to the buffer that can be temporarily used to
*                    store the requested data.
*   NumBytes       - Size of the given buffer used for checks
*
* Return value
*   Pointer to the start of the buffer used for storage.
*/
static const char * _UPnP_GetURLBase(char * pBuffer, unsigned NumBytes) {
#define URL_BASE_PREFIX  "http://"
  char * p;

  p = pBuffer;

  *p = '\0';  // Just to be on the safe if the buffer is too small
  strncpy(pBuffer, URL_BASE_PREFIX, NumBytes);
  p        += (sizeof(URL_BASE_PREFIX) - 1);
  NumBytes -= (sizeof(URL_BASE_PREFIX) - 1);
  IP_PrintIPAddr(p, IP_GetIPAddr(INTERFACE), NumBytes);
  return pBuffer;
}

/*********************************************************************
*
*       _UPnP_GetModelNumber
*
* Function description
*   This function copies the information needed for the ModelNumber parameter
*   into the given buffer and returns a pointer to the start of the buffer
*   for easy readable code.
*
* Parameters
*   pBuffer        - Pointer to the buffer that can be temporarily used to
*                    store the requested data.
*   NumBytes       - Size of the given buffer used for checks
*
* Return value
*   Pointer to the start of the buffer used for storage.
*/
static const char * _UPnP_GetModelNumber(char * pBuffer, unsigned NumBytes) {
  U8 aHWAddr[6];

  if (NumBytes <= 12) {
    *pBuffer = '\0';  // Just to be on the safe if the buffer is too small
  } else {
    IP_GetHWAddr(INTERFACE, aHWAddr, sizeof(aHWAddr));
     snprintf(pBuffer, NumBytes, "%02X%02X%02X%02X%02X%02X", aHWAddr[0], aHWAddr[1],
aHWAddr[2], aHWAddr[3], aHWAddr[4], aHWAddr[5]);
  }
  return pBuffer;
}

/*********************************************************************
*
*       _UPnP_GetSN
*
* Function description
*   This function copies the information needed for the SerialNumber parameter
*   into the given buffer and returns a pointer to the start of the buffer
```



```
*   for easy readable code.
*
* Parameters
*   pBuffer        - Pointer to the buffer that can be temporarily used to
*                    store the requested data.
*   NumBytes       - Size of the given buffer used for checks
*
* Return value
*   Pointer to the start of the buffer used for storage.
*/
static const char * _UPnP_GetSN(char * pBuffer, unsigned NumBytes) {
  U8 aHWAddr[6];

  if (NumBytes <= 12) {
    *pBuffer = '\0';  // Just to be on the safe if the buffer is too small
  } else {
    IP_GetHWAddr(INTERFACE, aHWAddr, sizeof(aHWAddr));
     snprintf(pBuffer, NumBytes, "%02X%02X%02X%02X%02X%02X", aHWAddr[0], aHWAddr[1],
aHWAddr[2], aHWAddr[3], aHWAddr[4], aHWAddr[5]);
  }
  return pBuffer;
}

/*********************************************************************
*
*       _UPnP_GetUDN
*
* Function description
*   This function copies the information needed for the UDN parameter
*   into the given buffer and returns a pointer to the start of the buffer
*   for easy readable code.
*
* Parameters
*   pBuffer        - Pointer to the buffer that can be temporarily used to
*                    store the requested data.
*   NumBytes       - Size of the given buffer used for checks
*
* Return value
*   Pointer to the start of the buffer used for storage.
*/
static const char * _UPnP_GetUDN(char * pBuffer, unsigned NumBytes) {
#define UDN_PREFIX "uuid:56F9C1D5-5083-4ee5-A6B3-"
  char * p;
  U8     aHWAddr[6];

  p = pBuffer;

  *pBuffer = '\0';  // Just to be on the safe if the buffer is too small
  strncpy(pBuffer, UDN_PREFIX, NumBytes);
  p        += (sizeof(UDN_PREFIX) - 1);
  NumBytes -= (sizeof(UDN_PREFIX) - 1);
  if (NumBytes > 12) {
    IP_GetHWAddr(INTERFACE, aHWAddr, sizeof(aHWAddr));
       snprintf(p,  NumBytes,  "%02X%02X%02X%02X%02X%02X",  aHWAddr[0],  aHWAddr[1],
aHWAddr[2], aHWAddr[3], aHWAddr[4], aHWAddr[5]);
  }
  return pBuffer;
}

/*********************************************************************
*
*       _UPnP_GetPresentationURL
*
* Function description
*   This function copies the information needed for the presentation URL parameter
*   into the given buffer and returns a pointer to the start of the buffer
*   for easy readable code.
```



```
*
* Parameters
*   pBuffer        - Pointer to the buffer that can be temporarily used to
*                    store the requested data.
*   NumBytes       - Size of the given buffer used for checks
*
* Return value
*   Pointer to the start of the buffer used for storage.
*/
static const char * _UPnP_GetPresentationURL(char * pBuffer, unsigned NumBytes) {
#define PRESENTATION_URL_PREFIX   "http://"
#define PRESENTATION_URL_POSTFIX  "/index.htm"
  char * p;
  int    i;

  p = pBuffer;

  *p = '\0';  // Just to be on the safe if the buffer is too small
  strncpy(pBuffer, PRESENTATION_URL_PREFIX, NumBytes);
  p        += (sizeof(PRESENTATION_URL_PREFIX) - 1);
  NumBytes -= (sizeof(PRESENTATION_URL_PREFIX) - 1);
  i = IP_PrintIPAddr(p, IP_GetIPAddr(INTERFACE), NumBytes);
  p        += i;
  NumBytes -= i;
  strncat(pBuffer, PRESENTATION_URL_POSTFIX, NumBytes);
  return pBuffer;
}

/*********************************************************************
*
*       _UPnP_GenerateSend_upnp_xml
*
* Function description
*   Send the content for the requested file using the callback provided.
*
* Parameters
*   pContextIn     - Send context of the connection processed for
*                    forwarding it to the callback used for output.
*   pf             - Function pointer to the callback that has to be
*                    for sending the content of the VFile.
*     pContextOut    - Out context of the connection processed.
*     pData          - Pointer to the data that will be sent
*     NumBytes       - Number of bytes to send from pData. If NumBytes
*                      is passed as 0 the send function will run a strlen()
*                      on pData expecting a string.
*
* Notes
*   (1) The data does not need to be sent in one call of the callback
*       routine. The data can be sent in blocks of data and will be
*       flushed out automatically at least once returning from this
*       routine.
*/
static void _UPnP_GenerateSend_upnp_xml(void * pContextIn, void (*pf) (void * pCon-
textOut, const char * pData, unsigned NumBytes)) {
  char ac[128];

  pf(pContextIn, "<?xml version=\"1.0\"?>\r\n"
                 "<root xmlns=\"urn:schemas-upnp-org:device-1-0\">\r\n"
                   "<specVersion>\r\n"
                     "<major>1</major>\r\n"
                     "<minor>0</minor>\r\n"
                                                                  "</specVer-
sion>\r\n"                                                  , 0);

                         pf(pContextIn,                                  "<URL-
Base>"                                                            , 0);
          pf(pContextIn,                        _UPnP_GetURLBase(ac,     sizeof(ac))
```



```
, 0);
                        pf(pContextIn,                                    "</URL-
Base>\r\n"                                                          , 0);

  pf(pContextIn,   "<device>\r\n"
                     "<deviceType>urn:schemas-upnp-org:device:Basic:1</device-
Type>\r\n"       , 0);
     pf(pContextIn,              "<friendlyName>"   UPNP_FRIENDLY_NAME   "</friend-
lyName>\r\n"               , 0);
     pf(pContextIn,              "<manufacturer>"   UPNP_MANUFACTURER   "</manufac-
turer>\r\n"                , 0);
   pf(pContextIn,         "<manufacturerURL>"  UPNP_MANUFACTURER_URL  "</manufacture-
rURL>\r\n"       , 0);
    pf(pContextIn,           "<modelDescription>"  UPNP_MODEL_DESC  "</modelDescrip-
tion>\r\n"           , 0);
       pf(pContextIn,                   "<modelName>"    UPNP_MODEL_NAME    "</model-
Name>\r\n"                         , 0);

                pf(pContextIn,                                              "<modelNum-
ber>"                                                             , 0);
       pf(pContextIn,                          _UPnP_GetModelNumber(ac,    sizeof(ac))
, 0);
                pf(pContextIn,                                              "</modelNum-
ber>\r\n"                                                         , 0);

        pf(pContextIn,                    "<modelURL>"    UPNP_MODEL_URL    "</mode-
lURL>\r\n"                              , 0);

                pf(pContextIn,                                              "<serialNum-
ber>"                                                             , 0);
         pf(pContextIn,                                _UPnP_GetSN(ac,     sizeof(ac))
, 0);
                pf(pContextIn,                                             "</serialNum-
ber>\r\n"                                                         , 0);

                  pf(pContextIn,                                                  "<UDN>"
, 0);
         pf(pContextIn,                                _UPnP_GetUDN(ac,     sizeof(ac))
, 0);
                pf(pContextIn,                                              "</UDN>\r\n"
, 0);

  pf(pContextIn,     "<serviceList>\r\n"
                       "<service>\r\n"
                        "<serviceType>urn:schemas-upnp-org:service:Dummy:1</service-
Type>\r\n"
                        "<serviceId>urn:upnp-org:serviceId:Dummy</serviceId>\r\n"
                        "<SCPDURL>/dummy.xml</SCPDURL>\r\n"
                        "<controlURL>/</controlURL>\r\n"
                        "<eventSubURL></eventSubURL>\r\n"
                      "</service>\r\n"
                                                                            "</service-
List>\r\n"                                                          , 0);

              pf(pContextIn,                                              "<presentation-
URL>"                                                           , 0);
      pf(pContextIn,                       _UPnP_GetPresentationURL(ac,   sizeof(ac))
, 0);
              pf(pContextIn,                                             "</presentation-
URL>\r\n"                                                       , 0);

  pf(pContextIn,   "</device>\r\n"
                                                                          "</root>"
, 0);
}
```


The callbacks for providing a virtual file using a VFile hook allow providing dynamically created content for every file requested from the web server as soon as possible. A file served from a VFile hook will not be processed further by the web server code.

```
/* Excerpt from OS_IP_WebserverUPnP.c */
/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _UPnP_CheckVFile
*
* Function description
*   Check if we have content that we can deliver for the requested
*   file using the VFile hook system.
*
* Parameters
*   sFileName      - Name of the file that is requested
*   pIndex         - Pointer to a variable that has to be filled with
*                    the index of the entry found in case of using a
*                    filename<=>content list.
*                    Alternative all comparisons can be done using the
*                    filename. In this case the index is meaningless
*                    and does not need to be returned by this routine.
*
* Return value
*   0              - We do not have content to send for this filename,
*                    fall back to the typical methods for retrieving
*                    a file from the web server.
*   1              - We have content that can be sent using the VFile
*                    hook system.
*/
static int _UPnP_CheckVFile(const char * sFileName, unsigned * pIndex) {
  unsigned i;

  //
  // Generated VFiles
  //
  if (strcmp(sFileName, "/upnp.xml") == 0) {
    return 1;
  }
  //
  // Static VFiles
  //
  for (i = 0; i < SEGGER_COUNTOF(_VFileList); i++) {
    if (strcmp(sFileName, _VFileList[i].sFileName) == 0) {
      *pIndex = i;
      return 1;
    }
  }
  return 0;
}

/*********************************************************************
*
*       _UPnP_SendVFile
*
* Function description
*   Send the content for the requested file using the callback provided.
*
* Parameters
*   pContextIn     - Send context of the connection processed for
```



```
*                     forwarding it to the callback used for output.
*   Index           - Index of the entry of a filename<=>content list
*                     if used. Alternative all comparisons can be done
*                     using the filename. In this case the index is
*                     meaningless. If using a filename<=>content list
*                     this is faster than searching again.
*   sFileName       - Name of the file that is requested. In case of
*                     working with the Index this is meaningless.
*   pf              - Function pointer to the callback that has to be
*                     for sending the content of the VFile.
*     pContextOut     - Out context of the connection processed.
*     pData           - Pointer to the data that will be sent
*     NumBytes        - Number of bytes to send from pData. If NumBytes
*                       is passed as 0 the send function will run a strlen()
*                       on pData expecting a string.
*/
static void _UPnP_SendVFile(void * pContextIn, unsigned Index, const char * sFile-
Name, void (*pf) (void * pContextOut, const char * pData, unsigned NumBytes)) {
  (void)sFileName;

  //
  // Generated VFiles
  //
  if (strcmp(sFileName, "/upnp.xml") == 0) {
    _UPnP_GenerateSend_upnp_xml(pContextIn, pf);
    return;
  }
  //
  // Static VFiles
  //
  pf(pContextIn, _VFileList[Index].pData, _VFileList[Index].NumBytes);
}

static WEBS_VFILE_APPLICATION _UPnP_VFileAPI = {
  _UPnP_CheckVFile,
  _UPnP_SendVFile
};
```

All that is needed to be added to your application in order to provide the necessary XML files through embOS/IP Web server and starting UPnP advertising are the following lines:

```
/* Excerpt from OS_IP_WebserverUPnP.c */
//
// Activate UPnP with VFile hook for needed XML files
//
IP_WEBS_AddVFileHook(&_UPnP_VFileHook, &_UPnP_VFileAPI);
IP_UPNP_Activate(INTERFACE, NULL);
```


11.5 API functions

Function	Description
IP_UPNP_Activate()	Activates UPnP advertisement of the target in the network.

Table 11.1: embOS/IP UPnP API function overview

11.5.1 IP_UPNP_Activate()

Description

Activates the UPnP server.

Prototype

```
void IP_UPNP_Activate( unsigned     IFace,
                       const char * acUDN );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based index of available network interfaces.
acUDN	[IN] String containing a unique descriptor name. (Optional, can be NULL.)

Table 11.2: IP_UPNP_Activate() parameter list

Additional infromation

If acUDN is NULL the unique descriptor name will be generated from the HW addr. of the interface.

11.6 UPnP resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the UPnP modules presented in the tables below have been measured on an ARM7 and a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

The pure size of the UPnP add-on has been measured as the size of the services provided may vary.

11.6.1 ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP UPnP	approximately 2.2Kbyte

Table 11.3: UPnP ROM usage ARM7

11.6.2 ROM usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

Addon	ROM
embOS/IP UPnP	approximately 2.0Kbyte

Table 11.4: UPnP ROM usage Cortex-M3

11.6.3 RAM usage

Addon	RAM
embOS/IP UPnP	approximately 170 bytes

Table 11.5: UPnP RAM usage

Chapter 12

VLAN

The embOS/IP implementation of VLAN which stand for Virtual LAN allows seperating your network into multiple networks without the need to seperate it physically. This chapter will show you how easily VLAN access can be setup on your target.

12.1 embOS/IP VLAN

The embOS/IP VLAN implementation allows a fast and easy implement of VLAN on your target. embOS/IP VLAN support supports a basic VLAN tag specifying only a VLAN ID.

12.2 Feature list

- Low memory footprint.
- Easy to implement.
- Software based solution without the need for a driver to support VLAN tags.

12.3 VLAN backgrounds

VLAN technology can be used to seperate multiple devices operating on the same physical network into completely seperated networks without seeing each other.

A typical usage would be to have 2 departments seperated from each other but using the same infrastructure such as a shared switch or router. Only devices using the same VLAN ID will be able to see each other.

For this to happen 4 bytes are added in front of the packet type field in the Ethernet frame pushing the original packet type field back by 4 bytes. The Ethernet frame will still be of a maximum length 1518 bytes including CRC what means that instead of a maximum of 1500 bytes that can be transferred the amount of bytes that can be transferred per Ethernet frame will shrink to 1496 bytes per packet. VLAN tagged packets are typically forwarded by any switch as they are as the type field has been simply replaced and in most cases only the destination MAC, source MAC and packet type is checked. In this case the packet is simply of an unknown protocol and will be forwarded by the switch.

The picture below shows the structure of an Ethernet frame once without using a VLAN tag and once with using a VLAN tag being assigned to VLAN ID #2.

Ethernet frame of max. 1518 bytes

Dest MAC	Src MAC	Packet Type	Packet Data
00:23:C7:FF:FF:FF	00:23:C7:FF:EE:EE	IP Packet 0x0800	Max. 1500 bytes data + 4 bytes CRC

Dest MAC	Src MAC	VLAN TAG		Packet Type	Packet Data
		TPI	16 bit TCI (12 bit VLAN ID)		
00:23:C7:FF:FF:FF	00:23:C7:FF:EE:EE	0x8100	VLAN ID #2 0x0002	IP Packet 0x0800	Max. 1496 bytes data + 4 bytes CRC

Ethernet frame of max. 1518 bytes

12.4 API functions

Function	Description
IP_VLAN_AddInterface()	Activates UPnP advertisement of the target in the network.

Table 12.1: embOS/IP VLAN API function overview

12.4.1 IP_VLAN_AddInterface()

Description

Adds a VLAN interface.

Prototype

```
int IP_VLAN_AddInterface( unsigned HWIFace,
                          U16      VLANId );
```

Parameter

Parameter	Description
HWIFace	[IN] Zero-based index of available network interfaces to be used as physical interface for the VLAN pseudo interface.
VLANId	[IN] 12 bit VLAN ID.

Table 12.2: IP_VLAN_AddInterface() parameter list

Return value

Zero-based index of the added VALN interface.

12.5 VLAN resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the VLAN modules presented in the tables below have been measured on an ARM7 and a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

12.5.1 ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP VLAN	approximately 1.2Kbyte

Table 12.3: VLAN ROM usage ARM7

12.5.2 ROM usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

Addon	ROM
embOS/IP VLAN	approximately 1.0Kbyte

Table 12.4: VLAN ROM usage Cortex-M3

12.5.3 RAM usage

Addon	RAM
embOS/IP VLAN	approximately 16 bytes

Table 12.5: VLAN RAM usage

Chapter 13

Network interface drivers

embOS/IP has been designed to cooperate with any kind of hardware. To use specific hardware with embOS/IP, a so-called network interface driver for that hardware is required. The network interface driver consists of basic functions for accessing the hardware and a global table that holds pointers to these functions.

13.1 General information

To use embOS/IP, a network interface driver matching the target hardware is required. The code size of a network interface driver depends on the hardware and is typically between 1 and 3 Kbytes. The driver handles both the MAC (media access control) unit as well as the PHY (Physical interface). We recommend using drivers written and tested by SEGGER. However, it is possible to write your own driver. This is explained in section *Writing your own driver* on page 248.

The driver interface has been designed to allow support of internal and external Ethernet controllers (EMACs). It also allows to take full advantage of hardware features such as MAC address filtering and checksum computation in hardware.

13.1.1 MAC address filtering

The stack passes a list of MAC addresses to the driver. The driver is responsible for making sure that all packets from all MAC addresses specified are passed to the stack. It can do so with "precise filtering" if the hardware has sufficient filters for the given number of MAC addresses. If more MAC addresses are passed to the driver than hardware filters are available, the driver can use a hash filter if available in hardware or switch to promiscuous mode.

This is a very flexible solution which allows making best use of the hardware filtering capabilities on all known Ethernet controllers. It also allows simple implementations to simply switch to promiscuous mode.

13.1.2 Checksum computation in hardware

When the interface is initialized, the stack queries the capabilities of the driver. If the hardware can compute IP, TCP, UDP, ICMP checksums, it can indicates this to the stack. In this case, the stack does not compute these checksums, improving throughput and reducing CPU load.

13.1.3 Ethernet CRC computation

Every Ethernet packet includes a 32-bit trailing CRC. In most cases, the Ethernet controller is capable of computing the CRC. The drivers take advantage of this. The CRC is computed in the driver only if the hardware does not support CRC computation.

13.2 Available network interface drivers

Network interface drivers are optional components to embOS/IP. The following network interface drivers are available:

Driver (Device)	Identifier
ATMEL AT91CAP9	IP_Driver_CAP9
ATMEL AT91RM9200	IP_Driver_AT91RM9200
ATMEL AT91SAM7X	IP_Driver_SAM7X
ATMEL AT91SAM9260	IP_Driver_SAM9260
ATMEL AT91SAM9263	IP_Driver_SAM9263
ATMEL AT91SAMG20	IP_Driver_SAM9G20
ATMEL AT91SAMG45	IP_Driver_SAMG45
ATMEL AT91SAM9XE	IP_Driver_SAM9XE
ATMEL AVR32UC	IP_Driver_AVR32UC
DAVICOM DM9000	IP_Driver_DM9000
FREESCALE ColdFire MCF5223x	IP_Driver_MCF5223x
FREESCALE ColdFire MCF5329	IP_Driver_MCF5329
NIOSII IFI GMACII EMAC	IP_Driver_GMACII
NIOSII MaCo-Engineering EMAC	IP_Driver_NIOSII_MaCo
NIOSII More than IP A2A bridge	IP_Driver_NIOSII_More10IP_A2A
NXP LPC17xx	IP_Driver_LPC17xx
NXP LPC2378 / LPC2478	IP_Driver_LPC24xx
NXP LPC32xx	IP_Driver_LPC32xx
RENESAS H8S2472	IP_Driver_H8S2472
RENESAS RX62N	IP_Driver_RX62N
RENESAS SH7670	IP_Driver_SH7670
RENESAS (NEC) V850JGH3	IP_Driver_V850JGH3
SMSC LAN9115 / LAN9215	IP_Driver_LAN9115
SMSC LAN9118	IP_Driver_LAN9118
SMSC LAN91C111	IP_Driver_LAN91C111
ST STM32F107 (Connectivity Line)	IP_Driver_STM32F107
ST STM32F207	IP_Driver_STM32F207
ST STR912	IP_Driver_STR912
TI (LUMINARY) LM3S6965	IP_Driver_LM3S6965
TI (LUMINARY) LM3S9B90	IP_Driver_LM3S9B90

Table 13.1: List of default network interface driver labels

To add a driver to embOS/IP, `IP_AddEtherInterface()` should be called with the proper identifier before the TCP/IP stack starts any transmission. Refer to *IP_AddEtherInterface()* on page 46 for detailed information.

13.2.1 ATMEL AT91CAP9

Atmel's CAP™ is a microcontroller-based system-on-chip platform with a Metal Programmable (MP) Block that allows the designer to add custom logic.

13.2.1.1 Supported hardware

The network interface driver for the AT91CAP9 can be used with every ATMEL AT91CAP9 target board. The driver has been tested on the following eval boards:

Tested evaluation boards
ATMEL CAP-DK

Table 13.2: List of tested eval boards

13.2.1.2 Configuring the driver

Adding the driver to embOS/IP

To add the driver, use `IP_AddEtherInterface()` with the driver identifier `IP_Driver_CAP9`. This function must be called from `IP_X_Config()`. Refer to *IP_AddEtherInterface()* on page 46 and *IP_X_Configure()* on page 256 for more information.

Example

```
void IP_X_Config(void) {
  int mtu;

  IP_AssignMemory(_aPool, sizeof(_aPool));    // Assigning memory
  IP_AddEtherInterface(&IP_Driver_CAP9);      // Add Ethernet driver
  IP_SetHWAddr("\x00\x22\xC7\xFF\xFF\xFF");   // MAC addr: Needs to be unique
                                              // for production units
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
  //
  // Set supported duplex modes
  // 10Mbit half duplex, 10Mbit full duplex, 100Mbit half duplex
  // and 100Mbit full duplex are supported.
  //
  IP_SetSupportedDuplexModes(0,  IP_PHY_MODE_10_HALF
                               | IP_PHY_MODE_10_FULL
                               | IP_PHY_MODE_100_HALF
                               | IP_PHY_MODE_100_FULL );
  IP_NI_ConfigPHYMode (0, 1);                 // Use RMII mode
  //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
  mtu = 1500;                                 // 576 is minimum acc. to RFC,
                                              // 1500 is max. for Ethernet
  IP_SetMTU(0, mtu);                          // Maximum Transmission Unit is
                                              // 1500 for ethernet by default
  IP_AddBuffers(12, 256);                     // Small buffers.
  IP_AddBuffers(8, mtu + 40 + 16);            // Big buffers. Size should be
                                              // mtu + 16 byte for ethernet header
                                              // (2 bytes type, 2*6 bytes MAC,
                                              // 2 bytes padding)
  IP_ConfTCPSpace(8 * (mtu-40), 8 * (mtu-40));
  //
  // Use DHCP client or define IP address, subnet mask,
  // gateway address and DNS server according to the
  // requirements of your application.
  //
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
  IP_SetWarnFilter(0xFFFFFFFF);               // 0xFFFFFFFF: Do not filter:
                                              // Output all warnings.
  IP_SetLogFilter(IP_MTYPE_INIT
                | IP_MTYPE_LINK_CHANGE
                | IP_MTYPE_DHCP);
}
```


13.2.1.3 Driver specific configuration functions

Function	Description
IP_NI_CAP9_ConfigNumRxBuffers()	Sets the number of Rx buffers.

Table 13.3: embOS/IP CAP9 driver specific function overview

13.2.1.3.1 IP_NI_CAP9_ConfigNumRxBuffers

Description

Sets the number of Rx buffers of the driver. This function has to be called in the configuration phase.

Prototype

```
void IP_NI_CAP9_ConfigNumRxBuffers( U16 NumRxBuffers );
```

Parameter

Parameter	Description
NumRxBuffers	[IN] The number of Rx buffers.

Table 13.4: IP_NI_CAP9_ConfigNumRxBuffers() parameter list

13.2.1.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 13.5: embOS/IP driver specific function overview

13.2.1.4.1 BSP_ETH_Init()

Description

This function is called from the network interface driver. It initializes the network interface. This function should be used to enable the ports which are connected to the network hardware. It is called from the driver during the initialization process.

Prototype

```
void BSP_ETH_Init( unsigned Unit );
```

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.

Table 13.6: BSP_ETH_Init() parameter list

Example

```
/* Excerpt of BSP.c for the ATMEL AT91CAP9 CAP-DK */

/*********************************************************************
*
*       BSP_ETH_Init()
*
*  Function description
*    This function is called from the network interface driver.
*    It initializes the network interface. This function should be used
*    to enable the ports which are connected to the network hardware.
*    It is called from the driver during the initialization process.
*/
void BSP_ETH_Init(unsigned Unit) {
  unsigned PinsA;
  unsigned v;

  _PMC_PCER          = (1 << _ID_EMAC_PORT);          // Enable clock for PIO
  _EMAC_PORT_PPUDR   = (1 << _EMAC_PORT_RXDV_BIT);    // Disable RXDV pullup,
                                                      // enter PHY normal mode
  //
  // Init PIO and perform a RESET of PHY since PHY
  //
  v              = 0
                 | (1 << _EMAC_PORT_RXDV_BIT)
                 ;
  _PIOB_PER      = v;
  _PIOB_OER      = v;
  _PIOB_CODR     = 0
                 | (1 << _EMAC_PORT_RXDV_BIT)
                 ;
  _PIOB_SODR     = 0
                 | (1 << 0)      // Isolate
                 ;
  //
  // Perform hardware reset using RESET pin of MCU
  //
  AT91C_RSTC_RMR = 0xA5000000 | AT91C_RSTC_ERSTL & (1 << 8);
  AT91C_RSTC_RCR = 0xA5000000 | AT91C_RSTC_EXTRST;
  while ((AT91C_RSTC_RSR & AT91C_RSTC_NRSTL) == 0); // Wait until RESET timer has
                                                    // expired (just a few ms)
  //
  // Init PIO Pins: EMAC is connected to specific lines of PIO
  //
  PinsA                = (1uL << 11)  // ETH_MDINTR
                       | (1uL << 21)  // ETXCK
                       | (1uL << 22)  // ERXDV
                       | (1uL << 23)  // ETX0
```



```
                          | (1uL << 24)  // ETX1
                          | (1uL << 25)  // ERX0
                          | (1uL << 26)  // ERX1
                          | (1uL << 27)  // ERXER
                          | (1uL << 28)  // ETXEN
                          | (1uL << 29)  // EMDC
                          | (1uL << 30)  // EMDIO
                         ;
  _EMAC_PORT_ASR  =   PinsA;            // Select peripheral A use
  _EMAC_PORT_PDR  =   PinsA;            // Disable GPIO mode,
                                        // select peripheral function
}
```

13.2.1.5 Additional information

None.

13.2.2 ATMEL AT91RM9200

The ATMEL AT919200 is based on the ARM920T processor. Its peripheral set includes USB Full Speed Host and Device Ports, 10/100 Base T Ethernet MAC, Multimedia Card Interface (MCI), three Synchronous Serial Controllers (SSC), four USARTs, Master/ Slave Serial Peripheral Interface (SPI), Timer Counters (TC) and Two Wire Interface (TWI), four 32-bit Parallel I/O Controllers and peripheral DMA channels.

13.2.2.1 Supported hardware

The network interface driver for the AT91RM9200 can be used with every ATMEL AT91RM9200 target board. The driver has been tested on the following eval board(s):

Tested evaluation boards
ATMEL AT91RM9200-EK

Table 13.7: List of tested eval boards

13.2.2.2 Configuring the driver

Adding the driver to embOS/IP

To add the driver, use `IP_AddEtherInterface()` with the driver identifier `IP_Driver_RM9200`. This function must be called from `IP_X_Config()`. Refer to *IP_AddEtherInterface()* on page 46 and *IP_X_Configure()* on page 256 for more information.

Example

```
void IP_X_Config(void) {
  int mtu;

  IP_AssignMemory(_aPool, sizeof(_aPool));      // Assigning memory
  IP_AddEtherInterface(&IP_Driver_AT91RM9200); // Add Ethernet driver
  IP_SetHWAddr("\x00\x22\xC7\xFF\xFF\xFF");     // MAC addr: Needs to be unique
                                                // for production units
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
  //
  // Set supported duplex modes
  // 10Mbit half duplex, 10Mbit full duplex, 100Mbit half duplex
  // and 100Mbit full duplex are supported.
  //
  IP_SetSupportedDuplexModes(0,  IP_PHY_MODE_10_HALF
                               | IP_PHY_MODE_10_FULL
                               | IP_PHY_MODE_100_HALF
                               | IP_PHY_MODE_100_FULL );
  IP_NI_ConfigPHYMode (0, 1);                 // Use RMII mode
  //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
  mtu = 1500;                                   // 576 is minimum acc. to RFC,
                                                // 1500 is max. for Ethernet
  IP_SetMTU(0, mtu);                            // Maximum Transmission Unit is
                                                // 1500 for ethernet by default
  IP_AddBuffers(12, 256);                       // Small buffers.
  IP_AddBuffers(8, mtu + 40 + 16);              // Big buffers. Size should be
                                                // mtu + 16 byte for ethernet header
                                                // (2 bytes type, 2*6 bytes MAC,
                                                // 2 bytes padding)
  IP_ConfTCPSpace(8 * (mtu-40), 8 * (mtu-40));
  //
  // Use DHCP client or define IP address, subnet mask,
  // gateway address and DNS server according to the
  // requirements of your application.
  //
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
  IP_SetWarnFilter(0xFFFFFFFF);                 // 0xFFFFFFFF: Do not filter:
                                                // Output all warnings.
```



```
  IP_SetLogFilter(IP_MTYPE_INIT
                  | IP_MTYPE_LINK_CHANGE
                  | IP_MTYPE_DHCP);
}
```

13.2.2.3 Driver specific configuration functions

Function	Description
IP_NI_AT91RM9200_ConfigNumRxBuffers()	Sets the number of Rx buffers.

Table 13.8: embOS/IP RM9200 driver specific function overview

13.2.2.3.1 IP_NI_AT91RM9200_ConfigNumRxBuffers

Description

Sets the number of Rx buffers of the driver. This function has to be called in the configuration phase.

Prototype

```
void IP_NI_AT91RM9200_ConfigNumRxBuffers( U16 NumRxBuffers );
```

Parameter

Parameter	Description
NumRxBuffers	[IN] The number of Rx buffers.

Table 13.9: IP_NI_RM9200_ConfigNumRxBuffers() parameter list

13.2.2.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 13.10: embOS/IP driver specific function overview

13.2.2.4.1 BSP_ETH_Init()

Description

This function is called from the network interface driver. It initializes the network interface. This function should be used to enable the ports which are connected to the network hardware. It is called from the driver during the initialization process.

Prototype

```
void BSP_ETH_Init( unsigned Unit );
```

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.

Table 13.11: BSP_ETH_Init() parameter list

Example

```
/* Excerpt of BSP.c for the ATMEL AT91RM9200-EK */

#define _PIOA_BASE_ADDR (0xFFFFF400UL)
#define _PMC_BASE_ADDR  (0xFFFFFC00UL)
#define _PIO_PUDR_OFF   (0x60)
#define _PIO_PUER_OFF   (0x64)
#define _PIO_ASR_OFF    (0x70)
#define _PIO_BSR_OFF    (0x74)
#define _PMC            (*(volatile unsigned int*)(_PMC_BASE_ADDR))
#define _PMC_PCER       (*(volatile unsigned int*)(_PMC_BASE_ADDR + 0x10))
#define _PMC_PCDR       (*(volatile unsigned int*)(_PMC_BASE_ADDR + 0x14))
#define _PIOA_ASR       (*(volatile unsigned int*)(_PIOA_BASE_ADDR + _PIO_ASR_OFF))
#define _PIOA_BSR       (*(volatile unsigned int*)(_PIOA_BASE_ADDR + _PIO_BSR_OFF))
#define _PIOA_PUDR      (*(volatile unsigned int*)(_PIOA_BASE_ADDR + _PIO_PUDR_OFF))
#define _PIOA_PUER      (*(volatile unsigned int*)(_PIOA_BASE_ADDR + _PIO_PUER_OFF))
#define _PIOA_ID        (2)    // Parallel IO Controller A
#define _PIOB_ID        (3)    // Parallel IO Controller B
#define _EMAC_ID        (24)   // EMAC

/*****************************************************************
*
*       BSP_ETH_Init()
*/
void BSP_ETH_Init(unsigned Unit) {
  unsigned int Pins;

  //
  // Initialize peripheral clock
  //
  _PMC_PCER  = (1 << _EMAC_ID);    // Ensure the clock for EMAC is enabled
  _PMC_PCER  = (1 << _PIOA_ID);    // Ensure the clock for PIOA is enabled
  _PIOA_PUDR = (1 << 11);          // Disable RXDV pullup, enter PHY normal mode
                                   // Note: the PHY has an internal pull-down
  _PIOA_PUER = (1 << 16);          // Enable Pull-Up on EMDIO pin
#ifdef RMII
  Pins  = ((unsigned int) (1 <<  7))
        | ((unsigned int) (1 <<  8))
        | ((unsigned int) (1 <<  9))
        | ((unsigned int) (1 << 10))
        | ((unsigned int) (1 << 11))
        | ((unsigned int) (1 << 12))
        | ((unsigned int) (1 << 13))
        | ((unsigned int) (1 << 14))
        | ((unsigned int) (1 << 15))
        | ((unsigned int) (1 << 16))
        ;
#else
```



```
#error "MII-mode not supported by AT91RM9200-EK"
#endif
  _PIOA_ASR  =   Pins;           // Select peripheral A use of the associated pins
  _PIOA_BSR  =   0;              // Select peripheral B, no peripheral B pins used
  _PIOA_PDR  =   Pins;           // Set peripheral control of the associated pins
}
```

13.2.2.5 Additional information

None.

13.2.3 ATMEL AT91SAM7X

The ATMEL AT91SAM7X's are flash microcontrollers with integrated Ethernet, USB and CAN interfaces, based on the 32-bit ARM7TDMI RISC processor.

13.2.3.1 Supported hardware

The network interface driver for the AT91SAM7X can be used with every ATMEL AT91SAM7X target board. The driver has been tested on the following eval boards:

Tested evaluation boards
ATMEL AT91SAM7X-EK
Olimex SAM7-EX256

Table 13.12: List of tested eval boards

13.2.3.2 Configuring the driver

Adding the driver to embOS/IP

To add the driver, use `IP_AddEtherInterface()` with the driver identifier `IP_Driver_SAM7X`. This function has to be called from `IP_X_Config()`. Refer to *IP_AddEtherInterface()* on page 46 and *IP_X_Configure()* on page 256 for more information.

Example

```
void IP_X_Config(void) {
  IP_AssignMemory(_aPool, sizeof(_aPool));    // Assigning memory
  IP_AddEtherInterface(&IP_Driver_SAM7X);     // Add Ethernet driver
  IP_SetHWAddr("\x00\x22\xC7\xFF\xFF\xFF");   // MAC addr: Needs to be unique
                                              // for production units
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
  //
  // Set supported duplex modes
  // 10Mbit half duplex, 10Mbit full duplex, 100Mbit half duplex
  // and 100Mbit full duplex are supported.
  //
  IP_SetSupportedDuplexModes(0, IP_PHY_MODE_10_HALF
                              | IP_PHY_MODE_10_FULL
                              | IP_PHY_MODE_100_HALF
                              | IP_PHY_MODE_100_FULL
                            );
  //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
  mtu = 1500;                                 // 576 is minimum acc. to RFC,
                                              // 1500 is max. for Ethernet
  IP_SetMTU(0, mtu);                          // Maximum Transmission Unit is
                                              // 1500 for ethernet by default
  IP_AddBuffers(12, 256);                     // Small buffers.
  IP_AddBuffers(6, mtu + 40 + 16);            // Big buffers. Size should be
                                              // mtu + 16 byte for ethernet header
                                              // (2 bytes type, 2*6 bytes MAC,
                                              // 2 bytes padding)
  IP_ConfTCPSpace(3 * (mtu-40), 3 * (mtu-40));
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
  IP_SetWarnFilter(0xFFFFFFFF);                // 0xFFFFFFFF: Do not filter:
                                               // Output all warnings.
  IP_SetLogFilter(IP_MTYPE_INIT
                | IP_MTYPE_LINK_CHANGE
                | IP_MTYPE_DHCP);
}
```


13.2.3.3 Driver specific configuration functions

Function	Description
IP_NI_SAM7X_ConfigNumRxBuffers()	Sets the number of Rx buffers.

Table 13.13: embOS/IP SAM7X driver specific function overview

13.2.3.3.1 IP_NI_SAM7X_ConfigNumRxBuffers()

Description

Sets the number of Rx buffers of the driver. This function has to be called in the configuration phase.

Prototype

```
void IP_NI_SAM7X_ConfigNumRxBuffers( U16 NumRxBuffers );
```

Parameter

Parameter	Description
NumRxBuffers	[IN] The number of Rx buffers.

Table 13.14: IP_NI_SAM7X_ConfigNumRxBuffers() parameter list

13.2.3.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 13.15: embOS/IP driver specific function overview

13.2.3.4.1 BSP_ETH_Init()

Description

This function is called from the network interface driver. It initializes the network interface. This function should be used to enable the ports which are connected to the network hardware. It is called from the driver during the initialization process.

Prototype

```
void BSP_ETH_Init( unsigned Unit );
```

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.

Table 13.16: BSP_ETH_Init() parameter list

Example

```
/* Excerpt from implementation for ATMEL AT91SAM7X-EK */

#define AT91C_PMC_PCER    (*(volatile unsigned*) 0xFFFFFC10)
#define AT91C_PIOB_PPUDR  (*(volatile unsigned*) 0xFFFFF660)
#define AT91C_PIOB_PER    (*(volatile unsigned*) 0xFFFFF600)
#define AT91C_PIOB_OER    (*(volatile unsigned*) 0xFFFFF610)
#define AT91C_PIOB_CODR   (*(volatile unsigned*) 0xFFFFF634)
#define AT91C_PIOB_SODR   (*(volatile unsigned*) 0xFFFFF630)
#define AT91C_PIOB_ODR    (*(volatile unsigned*) 0xFFFFF614)
#define AT91C_PIOB_PDR    (*(volatile unsigned*) 0xFFFFF604)
#define AT91C_RSTC_RMR    (*(volatile unsigned*) 0xFFFFFD08)
#define AT91C_PIOB_ASR    (*(volatile unsigned*) 0xFFFFF670)
#define AT91C_RSTC_RCR    (*(volatile unsigned*) 0xFFFFFD00)
#define AT91C_RSTC_ERSTL                         (0xF <<  8)
#define AT91C_RSTC_EXTRST                        (0x1 <<  3)
#define AT91C_RSTC_NRSTL                         (1UL << 16)

void BSP_ETH_Init(unsigned Unit) {
  unsigned v;

  AT91C_PMC_PCER                = (1 << _PIOB_ID); // Enable clock for PIOB
  AT91C_PIOB_PPUDR              = 1UL << 15;       // Disable RXDV pullup,
                                                   // enter PHY normal mode
  AT91C_PIOB_PPUDR              = 1UL << 16;
  //
  // Init PIO and perform a RESET of PHY since PHY
  //
  v                             =  0
                                | (1 << 0)
                                | (1 << 15)
                                | (1 << 16)
                                | (1 << 18)
                                ;

  AT91C_PIOB_PER                = v; // Entire lower 19 bits enabled
  AT91C_PIOB_OER                = v;
  AT91C_PIOB_CODR               = 0
                                | (1 << 7)      // 0: node mode, 1: repeater mode
                                | (1 << 15)     // 0: Normal mode, 1: test mode
```



```
                                  | (1 << 16)      // 0: MII
                                  | (1 << 18)      // 0: Power down
                                  ;

  AT91C_PIOB_SODR                 = 0
                                  | (1 << 0)       // Isolate
                                  ;

  //
  // Perform hardware reset using RESET pin of MCU
  //
  AT91C_RSTC_RMR = 0xA5000000 | AT91C_RSTC_ERSTL & (1 << 8);
  AT91C_RSTC_RCR = 0xA5000000 | AT91C_RSTC_EXTRST;
  while ((AT91C_RSTC_RSR & AT91C_RSTC_NRSTL) == 0); // Wait until RESET timer has
                                                    // expired
  //
  // Switch to peripheral functions
  //
  v = 0x3FFFF;                // Lower 18 bits are used for the peripheral
  AT91C_PIOB_ODR  = v;        // Entire lower 18 bits disabled
  AT91C_PIOB_ASR  = v;        // Select peripheral A use
  AT91C_PIOB_PDR  = v;        // Disable GPIO mode, select peripheral
}
```

13.2.3.5 Additional information

None.

13.2.4 ATMEL AT91SAM9260

The ATMEL AT91SAM9260 is based on the ARM926EJ-S™ processor. Its peripheral set includes USB Full Speed Host and Device interfaces, a 10/100 Base T Ethernet MAC, Image Sensor Interface, Multimedia Card Interface (MCI), Synchronous Serial Controllers (SSC), USARTs, Master/Slave Serial Peripheral Interfaces (SPI), a three-channel 16-bit Timer Counter (TC), a Two Wire Interface (TWI) and four-channel 10-bit ADC.

13.2.4.1 Supported hardware

The network interface driver for the AT91SAM9260 can be used with every ATMEL AT91SAM9260 target board. The driver has been tested on the following eval boards:

Tested evaluation boards
ATMEL AT91SAM9260

Table 13.17: List of tested eval boards

13.2.4.2 Configuring the driver

Adding the driver to embOS/IP

To add the driver, use `IP_AddEtherInterface()` with the driver identifier `IP_Driver_SAM9260`. This function must be called from `IP_X_Config()`. Refer to *IP_AddEtherInterface()* on page 46 and *IP_X_Configure()* on page 256 for more information.

Example

```
void IP_X_Config(void) {
  IP_AssignMemory(_aPool, sizeof(_aPool));    // Assigning memory
  IP_AddEtherInterface(&IP_Driver_SAM9260);   // Add Ethernet driver
  IP_SetHWAddr("\x00\x22\xC7\xFF\xFF\xFF");   // MAC addr: Needs to be unique
                                              // for production units
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
  //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
  IP_AddBuffers(50, 256);                     // Small buffers.
  IP_AddBuffers(50, 1536);                    // Big buffers. Size should be 1536 to
                                              // allow a full ether packet to fit.
  IP_ConfTCPSpace(16 * 1024, 16 * 1024);
  IP_SetWarnFilter(0xFFFFFFFF);               // 0xFFFFFFFF: Do not filter:
                                              // Output all warnings.
  IP_SetLogFilter(IP_MTYPE_INIT
                | IP_MTYPE_LINK_CHANGE
                | IP_MTYPE_DHCP);
}
```

13.2.4.3 Driver specific configuration functions

Function	Description
IP_NI_SAM9260_ConfigNumRxBuffers()	Sets the number of Rx buffers.

Table 13.18: embOS/IP SAM9260 driver specific function overview

13.2.4.3.1 IP_NI_SAM9260_ConfigNumRxBuffers

Description

Sets the number of Rx buffers of the driver. This function has to be called in the configuration phase.

Prototype

```
void IP_NI_SAM9260_ConfigNumRxBuffers( U16 NumRxBuffers );
```

Parameter

Parameter	Description
NumRxBuffers	[IN] The number of Rx buffers.

Table 13.19: IP_NI_SAM9260_ConfigNumRxBuffers() parameter list

13.2.4.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 13.20: embOS/IP driver specific function overview

13.2.4.4.1 BSP_ETH_Init()

Description

This function is called from the network interface driver. It initializes the network interface. This function should be used to enable the ports which are connected to the network hardware. It is called from the driver during the initialization process.

Prototype

```
void BSP_ETH_Init( unsigned Unit );
```

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.

Table 13.21: BSP_ETH_Init() parameter list

Example

```
/*********************************************************************
*
*       BSP_ETH_Init()
*
*  Function description
*    This function is called from the network interface driver.
*    It initializes the network interface. This function should be used
*    to enable the ports which are connected to the network hardware.
*    It is called from the driver during the initialization process.
*
*  Note:
*    (1) If your MAC is connected to the PHY via Media Independent
*        Interface (MII) change the macro _USE_RMII and call
*        IP_NI_ConfigPHYMode() from within IP_X_Config()
*        to change the default of driver.
*
*/
void BSP_ETH_Init(unsigned Unit) {
  unsigned PinsA;
  unsigned PinsB;

  PMC_PCER        = (1 << ID_EMAC_PORT);       // Enable clock for PIO
  EMAC_PORT_PPUDR = (1 << EMAC_PORT_RXDV_BIT); // Disable RXDV pullup,
                                               // enter PHY normal mode
#if _USE_RMII
  EMAC_PORT_PPUER = (1 << EMAC_PORT_RMII_BIT); // Enable Pullup  => Switch to RMII.
#else
  EMAC_PORT_PPUDR = (1 << EMAC_PORT_RMII_BIT); // Disable Pullup => Switch to MII.
#endif
  //
  // Power up PHY, may not be required, if set as hardwired option on target
  //
#ifdef EMAC_PORT_PWR_PHY_BIT
  EMAC_PORT_PER  = (1 << EMAC_PORT_PWR_PHY_BIT);
```



```
  EMAC_PORT_OER  = (1 << EMAC_PORT_PWR_PHY_BIT);
  EMAC_PORT_CODR = (1 << EMAC_PORT_PWR_PHY_BIT);
#endif
  //
  // Init PIO Pins: EMAC is connected to specific lines of PIO
  //
  PinsA                = (1uL << 12)
                       | (1uL << 13)
                       | (1uL << 14)
                       | (1uL << 15)
                       | (1uL << 16)
                       | (1uL << 17)
                       | (1uL << 18)
                       | (1uL << 19)
                       | (1uL << 20)
                       | (1uL << 21)
                       ;
  PinsB                = (1uL << 10)
                       | (1uL << 11)
                       | (1uL << 22)
                       | (1uL << 25)
                       | (1uL << 26)
                       | (1uL << 27)
                       | (1uL << 28)
                       | (1uL << 29)
                       ;
  EMAC_PORT_ASR = PinsA;          // Select peripheral A use
  EMAC_PORT_BSR = PinsB;          // Select peripheral B use
  EMAC_PORT_PDR = PinsA | PinsB; // Disable GPIO mode, select peripheral function
  //
  // Initialize priority of BUS MATRIX. EMAC needs highest priority for SDRAM access
  //
  MATRIX_SCFG3 = 0x01160030;     // Assign EMAC as default master, activate priority
arbitration, increase cycles
  MATRIX_PRAS3 = 0x00320000;     // Set Priority of EMAC to 3 (highest value)
}
```

13.2.4.5 Additional information

None.

13.2.5 DAVICOM DM9000/DM9000A

The Davicom DM9000 is a fully integrated single chip Fast Ethernet MAC controller with a generic processor interface, a 10/100M PHY and SRAM.

13.2.5.1 Supported hardware

The network interface driver for the Davicom DM9000 can be used with every target board which complies with the following:

- Davicom DM9000 is presented
- DM 9000 is connected to the data/address bus; data bus is 16-bits wide
- INT pin connected to CPU in a way which allows generating interrupts

The driver has been tested on the following eval boards:

Tested evaluation boards
ATMEL AT91SAM9261-EK

Table 13.22: List of tested eval boards

13.2.5.2 Configuring the driver

Adding the driver to embOS/IP

To add the driver, use `IP_AddEtherInterface()` with the driver identifier `IP_Driver_DM9000`. This function must be called from within `IP_X_Config()`. Refer to *IP_AddEtherInterface()* on page 46 and *IP_X_Configure()* on page 256 for more information.

Example

```
void IP_X_Config(void) {
  IP_AssignMemory(_aPool, sizeof(_aPool));     // Assigning memory
  IP_AddEtherInterface(&IP_Driver_DM9000);     // Add Ethernet driver
  IP_NI_DM9000_ConfigAddr(0, (void*) (0x30000000), (void*) (0x30000000 + 0x04));
  IP_NI_ConfigPoll(0);                         // No ISR routine
  IP_SetHWAddr("\x00\x22\x33\x44\x55\x66");    // MAC addr: Needs to be unique
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
  //
  // Run-time configure buffers. The default setup will do for most cases.
  //
  IP_AddBuffers(12, 256);                      // Small buffers.
  IP_AddBuffers(12, 1536);                     // Big buffers. Size should be 1536 to
                                               // allow a full ether packet to fit.
  IP_ConfTCPSpace(6 * 1024, 4 * 1024);
  IP_SetWarnFilter(0xFFFFFFFF);                // 0xFFFFFFFF: Do not filter:
                                               // Output all warnings.
  IP_SetLogFilter(IP_MTYPE_INIT
                | IP_MTYPE_LINK_CHANGE
                | IP_MTYPE_DHCP );
}
```


13.2.5.3 Driver-specific configuration functions

Function	Description
IP_NI_DM9000_ConfigAddr()	Sets the base address for commands and data register.
IP_NI_DM9000_ISR_Handler()	Interrupt service routine for the network interface.

Table 13.23: embOS/IP DM9000 driver-specific function overview

13.2.5.3.1 IP_NI_DM9000_ConfigAddr()

Description

Sets the base address (for command) and data address.

Prototype

```
void IP_NI_DM9000_ConfigAddr( unsigned   Unit,
                              void     * pBase,
                              void     * pValue );
```

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.
pBase	[IN] Pointer to the control register of the MAC.
pValue	[IN] Pointer to the data register of the MAC.

Table 13.24: IP_NI_DM9000_ConfigAddr() parameter list

Additional information

This function must be called from within IP_X_Config. Refer to *IP_X_Configure()* on page 256 for detailed information.

13.2.5.3.2 IP_NI_DM9000_ISR_Handler()

Description

This is the interrupt service routine for the network interface (EMAC). It handles all interrupts (Rx, Tx, Error).

Prototype

```
void IP_NI_DM9000_ISR_Handler( unsigned Unit );
```

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.

Table 13.25: IP_NI_DM9000_ISR_Handler() parameter list

13.2.5.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 13.26: embOS/IP driver specific function overview

13.2.5.4.1 BSP_ETH_Init()

Description

This function is called from the network interface driver. It initializes the network interface. This function should be used to enable the ports which are connected to the network hardware. It is called from the driver during the initialization process.

Prototype

```
void BSP_ETH_Init( unsigned Unit );
```

13.2.5.5 Additinal information

None.

13.2.6 FREESCALE ColdFire MCF5329

13.2.6.1 Supported hardware

The network interface driver for the ColdFire MCF5329 MCU can be used with every target board. The driver has been tested on the following eval boards:

Tested evaluation boards
LOGICPD ZOOM COLDFIRE SDK with MCF5329 Fire Engine

Table 13.27: List of tested eval boards

13.2.6.2 Configuring the driver

Adding the driver to embOS/IP

To add the driver, use `IP_AddEtherInterface()` with the driver identifier `IP_Driver_MCF5329`. This function must be called from `IP_X_Config()`. Refer to *IP_AddEtherInterface()* on page 46 and *IP_X_Configure()* on page 256 for more information.

Example

```
/* Sample implementation taken from the configuration for the ColdFire MCF5329 */

#define ALLOC_SIZE                0xA000      // Size of memory dedicated
                                              // to the stack in bytes
U32 _aPool[ALLOC_SIZE / 4];                   // This is the memory area used
                                               // by the stack.


/*************************************************************************
*
*       IP_X_Config
*/
void IP_X_Config(void) {
  IP_AssignMemory(_aPool, sizeof(_aPool));    // Assigning memory
  IP_AddEtherInterface(&IP_Driver_MCF5329);   // Add ethernet driver
  IP_SetHWAddr((const unsigned char *)"\x00\x22\xC7\xFF\xFF\xFF");
  //
  // Use DHCP client or define IP address, subnet mask,
  // gateway address and DNS server according to the
  // requirements of your application.
  //
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
//  IP_SetAddrMask(0xC0A805E6, 0xFFFF0000);     // Assign IP addr. and subnet mask
//  IP_SetGWAddr(0, 0xC0A80201);                // Set gateway address
//  IP_DNS_SetServer(0xCC98B84C);               // Set DNS server address,
                                                // for example 204.152.184.76
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
  //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
  IP_AddBuffers(12, 256);                    // Small buffers.
  IP_AddBuffers(10, 1536);                   // Big buffers.
  IP_ConfTCPSpace(4 * 1024, 4 * 1024);       // Define the TCP Tx and Rx window size
  //
  // Define log and warn filter
  //
  IP_SetWarnFilter(0xFFFFFFFF);
  IP_SetLogFilter(IP_MTYPE_INIT
```



```
                    | IP_MTYPE_LINK_CHANGE
                    | IP_MTYPE_DHCP
                    );
}
```

13.2.6.3 Driver-specific configuration functions

None.

13.2.6.4 Required BSP functions

None.

13.2.6.5 Additional information

None.

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.

Table 13.28: BSP_ETH_Init() parameter list

Example

```
/* Excerpt from implementation for the ATMEL AT91SAM9261-EK */

#define _PIOC_ID    (4)
#define _PMC_PCER   (*(volatile unsigned int*) 0xFFFFF810)
#define _PIOC_PER   (*(volatile unsigned int*) 0xFFFFFC00)
#define _PIOC_ODR   (*(volatile unsigned int*) 0xFFFFFC14)
#define _PIOC_OER   (*(volatile unsigned int*) 0xFFFFFC10)
#define _PIOC_SODR  (*(volatile unsigned int*) 0xFFFFFC30)
#define _PIOC_CODR  (*(volatile unsigned int*) 0xFFFFFC34)

/*********************************************************************
*
*       BSP_ETH_Init()
*/
void BSP_ETH_Init(unsigned Unit) {
  int i;
  _PMC_PCER   |= (1 << _PIOC_ID);       // Enable peripheral clock
  _PIOC_PER   = (1 << 10) | (1 << 11); // Enable Ports for RESET and Interrupt
  _PIOC_OER   = (1 << 10);              // Switch RESET to output mode
  _PIOC_ODR   = (1 << 11);              // Switch Interrupt to output mode
  //
  // Activate & deactivate RESET of Ethernet controller.
  // We do this in a loop to allow sufficient time for Controller to get out of RESET
  //
  for (i = 0; i < 1000; i++) {
    _PIOC_SODR  = (1 << 10);             // Activate RESET
  }
  for (i = 0; i < 1000; i++) {
    _PIOC_CODR  = (1 << 10);             // Deactivate RESET
  }
}
```

13.2.6.6 Additional information

None.

13.2.7 NXP LPC17xx

The NXP LPC17xx MCUs are flash microcontrollers with integrated Ethernet, USB and CAN interfaces, based on the 32-bit Cortex-M3 processor.

13.2.7.1 Supported hardware

The network interface driver for the NXP 17xx can be used with every NXP LPC17xx target board. The driver has been tested on the following eval boards:

Tested evaluation boards
KEIL MCB1760
IAR LPC1768-SK
EmbeddedArtists LPC1788

Table 13.29: List of tested eval boards

13.2.7.2 Configuring the driver

Adding the driver to embOS/IP

To add the driver, use `IP_AddEtherInterface()` with the driver identifier `IP_Driver_LPC24xx`. This function must be called from `IP_X_Config()`. Refer to *IP_AddEtherInterface()* on page 46 and *IP_X_Configure()* on page 256 for more information.

Example

```
/* Sample implementation taken from the configuration for the NXP LPC2468 */

/*********************************************************************
*
*       IP_X_Config
*
*  Function description
*    This function is called by the IP stack during IP_Init().
*/
void IP_X_Config(void) {
  IP_AssignMemory(_aPool, sizeof(_aPool));    // Assigning memory
  IP_AddEtherInterface(&IP_Driver_LPC17xx);   // Add ethernet driver
  IP_SetHWAddr("\x00\x22\x33\x44\x55\x66");   // MAC addr: Needs to be unique
                                              // for production units
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
  //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
  IP_AddBuffers(6, 256);                      // Small buffers.
  IP_AddBuffers(8, 1536);                     // Big buffers. Size should be 1536
                                              // to allow a full ether packet to fit.
  IP_ConfTCPSpace(6 * 1024, 6 * 1024);
  IP_SetWarnFilter(0xFFFFFFFF);               // Do not filter: Output all warnings.
  IP_SetLogFilter(IP_MTYPE_INIT
                | IP_MTYPE_LINK_CHANGE
                );
}
```

13.2.7.3 Driver-specific configuration functions

None.

13.2.7.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 13.30: embOS/IP driver specific function overview

13.2.7.4.1 BSP_ETH_Init()

Description

This function is called from the network interface driver. It initializes the network interface. This function should be used to enable the ports which are connected to the network hardware. It is called from the driver during the initialization process.

Prototype

```
void BSP_ETH_Init( unsigned Unit );
```

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.

Table 13.31: BSP_ETH_Init() parameter list

Example

```
/* Sample implementation for NXP LPC2468 */

#define PINSEL2       *(volatile unsigned long *)(0xE002C008)
#define PINSEL3       *(volatile unsigned long *)(0xE002C00C)

/*********************************************************************
*
*       ETH_Init
*/
void BSP_ETH_Init(unsigned Unit) {
  /*-------------------------------------------------------------------------
   * write to PINSEL2/3 to select the PHY functions on P1[17:0]
   *-------------------------------------------------------------------------*/
  /* P1.6, ENET-TX_CLK, has to be set for EMAC to address a BUG in
     the rev"xx-X" or "xx-Y" silicon(see errata). On the new rev.(xxAY, released
     on 06/22/2007), P1.6 should NOT be set. */
  if (MAC_MODULEID == 0x39022000) {     // Older chip ?
    PINSEL2 = 0x50151105;  /* Selects P1[0,1,4,6,8,9,10,14,15] */
  } else {
    PINSEL2 = 0x50150105;  /* Selects P1[0,1,4,8,9,10,14,15] */
  }
  PINSEL3 = (PINSEL3 & ~0x0000000f) | 0x5;
}
```

13.2.7.5 Additional information

None.

13.2.8 NXP LPC23xx / 24xx

The NXP LPC23xx and LPC24xx MCU families are flash microcontrollers with integrated Ethernet, USB and CAN interfaces, based on the 32-bit ARM7TDMI-S RISC processor.

13.2.8.1 Supported hardware

The network interface driver for the NXP LPC23xx and LPC24xx MCUs can be used with every NXP LPC23xx/LPC24xx target board. The driver has been tested on the following eval boards:

Tested evaluation boards
KEIL MCB2300
IAR LPC2468 V1.0
EmbeddedArtists LPC2468

Table 13.32: List of tested eval boards

13.2.8.2 Configuring the driver

Adding the driver to embOS/IP

To add the driver, use `IP_AddEtherInterface()` with the driver identifier `IP_Driver_LPC24xx`. This function must be called from `IP_X_Config()`. Refer to *IP_AddEtherInterface()* on page 46 and *IP_X_Configure()* on page 256 for more information.

Example

```
/* Sample implementation taken from the configuration for the NXP LPC2468 */

/*********************************************************************
*
*       IP_X_Config
*
*  Function description
*    This function is called by the IP stack during IP_Init().
*/
void IP_X_Config(void) {
  IP_AssignMemory(_aPool, sizeof(_aPool));    // Assigning memory
  IP_AddEtherInterface(&IP_Driver_LPC24xx);   // Add ethernet driver
  IP_SetHWAddr("\x00\x22\x33\x44\x55\x66");   // MAC addr: Needs to be unique
                                              // for production units
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
  //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
  IP_AddBuffers(6, 256);                      // Small buffers.
  IP_AddBuffers(8, 1536);                     // Big buffers. Size should be 1536
                                              // to allow a full ether packet to fit.
  IP_ConfTCPSpace(6 * 1024, 6 * 1024);
  IP_SetWarnFilter(0xFFFFFFFF);               // Do not filter: Output all warnings.
  IP_SetLogFilter(IP_MTYPE_INIT
                | IP_MTYPE_LINK_CHANGE
                );
}
```


13.2.8.3 Driver-specific configuration functions

None.

13.2.8.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 13.33: embOS/IP driver specific function overview

13.2.8.4.1 BSP_ETH_Init()

Description

This function is called from the network interface driver. It initializes the network interface. This function should be used to enable the ports which are connected to the network hardware. It is called from the driver during the initialization process.

Prototype

```
void BSP_ETH_Init( unsigned Unit );
```

Parameter

Parameter	Description
Unit	[IN] Zero-based index of available network interfaces.

Table 13.34: BSP_ETH_Init() parameter list

Example

```
/* Sample implementation for NXP LPC2468 */

#define PINSEL2       *(volatile unsigned long *)(0xE002C008)
#define PINSEL3       *(volatile unsigned long *)(0xE002C00C)

/*********************************************************************
*
*       ETH_Init
*/
void BSP_ETH_Init(unsigned Unit) {
  /*-------------------------------------------------------------------------
   * write to PINSEL2/3 to select the PHY functions on P1[17:0]
   *-------------------------------------------------------------------------*/
  /* P1.6, ENET-TX_CLK, has to be set for EMAC to address a BUG in
     the rev"xx-X" or "xx-Y" silicon(see errata). On the new rev.(xxAY, released
     on 06/22/2007), P1.6 should NOT be set. */
  if (MAC_MODULEID == 0x39022000) {     // Older chip ?
    PINSEL2 = 0x50151105;  /* Selects P1[0,1,4,6,8,9,10,14,15] */
  } else {
    PINSEL2 = 0x50150105;  /* Selects P1[0,1,4,8,9,10,14,15] */
  }
  PINSEL3 = (PINSEL3 & ~0x0000000f) | 0x5;
}
```

13.2.8.5 Additional information

None.

13.2.9 ST STR912

The ST STR912 is based on the ARM966E-S™ processor. It is a flash microcontroller with integrated Ethernet, USB and CAN interfaces, AC Motor Control, 4 Timers, ADC, RTC, and DMA.

13.2.9.1 Supported hardware

The network interface driver for the STR912 can be used with every target ST STR912 target board. The driver has been tested on the following eval boards:

Tested evaluation boards
IAR STR912FA development board

Table 13.35: List of tested eval boards

13.2.9.2 Configuring the driver

Adding the driver to embOS/IP

To add the driver, use `IP_AddEtherInterface()` with the driver identifier `IP_Driver_STR912`. This function must be called from `IP_X_Config()`. Refer to *IP_AddEtherInterface()* on page 46 and *IP_X_Configure()* on page 256 for more information.

Example

```
/* Sample implementation taken from the configuration for the ST STR912 */

void IP_X_Config(void) {
  IP_AssignMemory(_aPool, sizeof(_aPool));     // Assigning memory
  IP_AddEtherInterface(&IP_Driver_STR912);     // Add Ethernet driver
  IP_SetHWAddr("\x00\x22\x33\x44\x55\x66");    // MAC addr: Needs to be unique
                                               // for production units
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
   //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
  IP_AddBuffers(20, 256);                      // Small buffers.
  IP_AddBuffers(12, 1536);                     // Big buffers. Size should be 1536
                                               // to allow a full ether packet to fit.
  IP_ConfTCPSpace(8 * 1024, 8 * 1024);
  IP_SetWarnFilter(0xFFFFFFFF);                // 0xFFFFFFFF: Do not filter:
                                               // Output all warnings.
  IP_SetLogFilter(IP_MTYPE_INIT
                | IP_MTYPE_LINK_CHANGE
                | IP_MTYPE_DHCP);
}
```

13.2.9.3 Driver-specific configuration functions

None.

13.2.9.4 Required BSP functions

None.

13.2.9.5 Additional information

None.

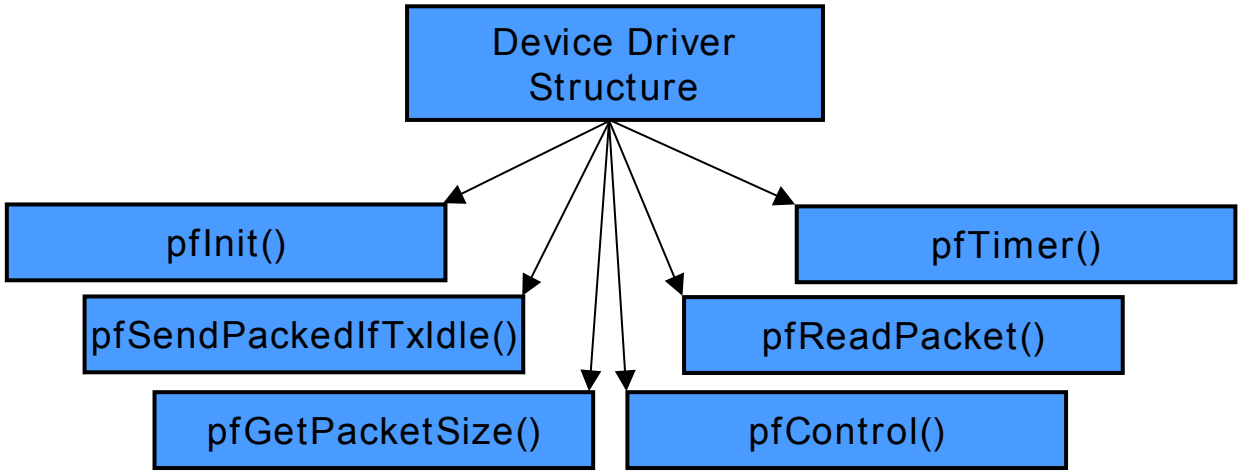
13.3 Writing your own driver

If you are going to use embOS/IP with your own hardware, you may have to write your own network interface driver. This section describes which functions are required and how to integrate your own network interface driver into embOS/IP.

Note: We strongly recommend contacting SEGGER if you need to have a driver for a particular piece of hardware which is not yet supported. Writing a driver is a difficult task which requires a thorough understanding of Ethernet, MAC, and PHY.

13.3.1 Network interface driver structure

embOS/IP uses a simple structure with function pointers to call the appropriate driver function for a device. Use the supplied template `IP_NI_Template.c` for the implementation.



Data structure

```
typedef struct IP_HW_DRIVER {
  int   (*pfInit)           ( unsigned Unit );
  int   (*pfSendPacket)     ( unsigned Unit );
  int   (*pfGetPacketSize)  ( unsigned Unit );
  int   (*pfReadPacket)     ( unsigned Unit, U8 * pDest, unsigned NumBytes );
  void  (*pfTimer)          ( unsigned Unit );
  int   (*pfControl)        ( unsigned Unit, int Cmd, void * p );
} IP_HW_DRIVER;
```

Elements of IP_HW_DRIVER

Element	Meaning
pfInit	Pointer to the initialization function.
pfSendPacket	Pointer to the send packet function.
pfGetPacketSize	Pointer to the get packet size function.
pfReadPacket	Pointer to the read packet function.
pfTimer	Optional: Pointer to the timer function. The routine is called from the stack periodically.
pfControl	Pointer to the control function.

Table 13.36: IP_HW_DRIVER - List of structure member variables

Example

```
/* Sample implementation taken from the driver for the ATMEL AT91SAM7X */

/*********************************************************************
*
*       Driver API Table
*
*********************************************************************
*/
const IP_HW_DRIVER IP_Driver_SAM7X = {
  _Init,
  _SendPacketIfTxIdle,
  _GetPacketSize,
  _ReadPacket,
  _Timer,
  _Control
};
```


13.3.2 Device driver functions

This section provides descriptions of the network interface driver functions required by embOS/IP. Note that the names used for these functions are not really relevant for embOS/IP because the stack accesses them through a structure of function pointers.

Function	Description
pfControl()	This function is used to implement additional driver specific control functions. It can be empty.
pfInit()	General initialization function of the driver.
pfGetPacketSize()	Reads buffer descriptors to find out if a packet has been received.
pfReadPacket()	Reads the first packet in the buffer.
pfSendPacketIfTxIdle()	Send the next packet in the send queue if transmitter is idle.
pfTimer()	Timer function called by the networking task, IP_Task(), once per second.

Table 13.37: embOS/IP network interface driver functions

13.3.3 Driver template

The driver template `IP_NI_Template.c` is supplied in the folder `Sample\Driver\Template\`.

Example

```
/*********************************************************************
*                SEGGER MICROCONTROLLER SYSTEME GmbH                 *
*       Solutions for real time microcontroller applications         *
**********************************************************************
*                                                                    *
*       (C) 2007 - 2008   SEGGER Microcontroller Systeme GmbH        *
*                                                                    *
*       www.segger.com     Support: support@segger.com               *
*                                                                    *
**********************************************************************
*                                                                    *
*       TCP/IP stack for embedded applications                       *
*                                                                    *
**********************************************************************
----------------------------------------------------------------------
File    : IP_NI_Template.c
Purpose : Network interface driver template
--------  END-OF-HEADER  ---------------------------------------------
*/

#include  "IP_Int.h"

/*********************************************************************
*
*       _SetFilter
*
*  Function description
*    Sets the MAC filter(s)
*    The stack tells the driver which addresses should go thru  the filter.
*    The number of addresses can generally be unlimited.
*    In most cases, only one address is set.
*    However, if the NI is in multiple nets at the same time or if multicast is used,
*    multiple addresses can be set.
*
*  Notes
*    (1) Procedure
*        In general, precise filtering is used as far as supported by the hardware.
*        If the more addresses need to be filtered than precise address filters are
*        available, then the hash filter is used.
*        Alternativly, the MAC can be switched to promiscuous mode for simple
*        implementations.
*/
static int _SetFilter(IP_NI_CMD_SET_FILTER_DATA * pFilter) {
  U32 v;
  U32 w;
  unsigned i;
  unsigned NumAddr;
  const U8 * pAddrData;

  NumAddr = pFilter->NumAddr;
  for (i = 0; i < NumAddr; i++) {
    pAddrData    = *(&pFilter->pHWAddr + i);


  }
  return 0;     // O.K.
}


/*********************************************************************
*
*       _SendPacket
*
*  Function description
*    Send the next packet in the send queue.
*    Function is called from 2 places:
*    - from a task via  pfSendPacketIfTxIdle() in Driver structure
*    - from ISR when Tx is completed (TxInterrupt)
*/
static int  _SendPacket(void) {
  U32      v;
  void   * pPacket;
  unsigned NumBytes;
```



```
  IP_GetNextOutPacket(&pPacket, &NumBytes);       // Get information about next
                                                  // packet in the Queue. 0
                                                  // means no packet in queue
  if (NumBytes == 0) {
    return 0;
  }
  IP_LOG((IP_MTYPE_DRIVER, "DRIVER: Sending packet: %d bytes", NumBytes));

  //
  // Start send
  //

  return 0;
}


/*********************************************************************
*
*       _ISR_Handler
*
*  Function description
*    This is the interrupt service routine for the NI (EMAC).
*    It handles all interrupts (Rx, Tx, Error).
*
*/
static void  _ISR_Handler(void) {
}


/*********************************************************************
*
*       _Init
*
*  Function description
*    General init function of the driver.
*    Called by the stack in the init phase before any other driver function.
*/
static  int  _Init(unsigned Unit) {
  int r;

  r = _PHY_Init(Unit);                            // Configure the PHY
  if (r) {
    return 1;
  }
  //
  // TBD
  //
  return 0;
}


/*********************************************************************
*
*       _SendPacketIfTxIdle
*
*  Function description
*    Send the next packet in the send queue if transmitter is idle.
*    If transmitter is busy, nothing is done since the next packet is sent
*    automatically with Tx-interrupt.
*    Function is called from a task via function pointer in in driver structure.
*/
static int  _SendPacketIfTxIdle(unsigned Unit) {
  //
  // TBD
  //
  return 0;
}


/*********************************************************************
*
*       _GetPacketSize()
*
*  Function description
*    Reads buffer descriptors in order to find out if a packet has been received.
*    Different error conditions are checked and handled.
*    Function is called from a task via function pointer in driver structure.
*
*  Return value
*    Number of buffers used for the next packet.
```



```
*    0 if no complete packet is available.
*/
static int _GetPacketSize(unsigned Unit) {
  //
  // TBD
  //
  return 0;
}


/*********************************************************************
*
*       _ReadPacket
*
*  Function description
*    Reads the first packet into the buffer.
*    NumBytes must be the correct number of bytes as retrieved by _GetPacketSize();
*    Function is called from a task via function pointer in driver structure.
*
*/
static int _ReadPacket(unsigned Unit, U8 *pDest, unsigned NumBytes) {
  //
  // TBD
  //
  return 0;
}


/*********************************************************************
*
*       _Timer
*
*  Function description
*    Timer function called by the Net task once per second.
*    Function is called from a task via function pointer in driver structure.
*/
static void _Timer(unsigned Unit) {
//  _UpdateLinkState();
}


/*********************************************************************
*
*       _Control
*
*  Function description
*    Control function for various purposes.
*    Function is called from a task via function pointer in driver structure.
*
*  Return value
*    -1:   Command is not supported
*  !=-1:   Command supported. Typically 0 means success,
*          but can also be a return value.
*/
static int _Control(unsigned Unit, int Cmd, void * p) {
  switch (Cmd) {
  case IP_NI_CMD_SET_FILTER:
    return _SetFilter((IP_NI_CMD_SET_FILTER_DATA*)p);
  case IP_NI_CMD_SET_BPRESSURE:
    //
    // TBD: Enable back pressure (if supported) and change return value to 0
    //
    break;
  case IP_NI_CMD_CLR_BPRESSURE:
    //
    // TBD: Disable back pressure (if supported) and change return value to 0
    //
    break;
  case IP_NI_CMD_GET_MAC_ADDR:
    break;
  case IP_NI_CMD_GET_CAPS:
    //
    // TBD: Retrieves the capabilites, which are a logical-or combination of
    // the IP_NI_CAPS (if any)
    //
    // {
    // int v;
    //
    // v = 0
    //   | IP_NI_CAPS_WRITE_IP_CHKSUM    // Driver capable of inserting the
                                         // IP-checksum into an outgoing packet?
```



```
    //   | IP_NI_CAPS_WRITE_UDP_CHKSUM   // Driver capable of inserting the
                                         // UDP-checksum into an outgoing packet?
    //   | IP_NI_CAPS_WRITE_TCP_CHKSUM   // Driver capable of inserting the
                                         // TCP-checksum into an outgoing packet?
    //   | IP_NI_CAPS_WRITE_ICMP_CHKSUM  // Driver capable of inserting the
                                         // ICMP-checksum into an outgoing packet?
    //   | IP_NI_CAPS_CHECK_IP_CHKSUM    // Driver capable of computing and
                                         // comparing the IP-checksum of
                                         // incoming packets?
    //   | IP_NI_CAPS_CHECK_UDP_CHKSUM   // Driver capable of computing and
                                         // comparing the UDP-checksum of an
                                         // incoming packet?
    //   | IP_NI_CAPS_CHECK_TCP_CHKSUM   // Driver capable of computing
                                         // and comparing the TCP-checksum of
                                         // an incoming packet?
    //   | IP_NI_CAPS_CHECK_ICMP_CHKSUM  // Driver capable of computing
                                         // and comparing the ICMP-checksum of
                                         // an incoming packet?
    // }
    // return v;
    break;
  case IP_NI_CMD_POLL:
    //
    //  Poll MAC (typically once per ms) in cases where MAC does not
    //  trigger an interrupt.
    //
    break;
  default:
    ;
  }
  return -1;
}

/*********************************************************************
*
*       Public API struct
*
*  This is the only public part of the driver.
*  All driver functions are called indirectly via this structure
*
*/
const IP_HW_DRIVER IP_Driver_Template = {
  _Init,
  _SendPacketIfTxIdle,
  _GetPacketSize,
  _ReadPacket,
  _Timer,
  _Control
};

/************************** End of file ****************************/
```


Chapter 14

Configuring embOS/IP

embOS/IP can be used without changing any of the compile-time flags. All compile-time configuration flags are preconfigured with valid values, which match the requirements of most applications. Network interface drivers can be added at runtime.

The default configuration of embOS/IP can be changed via compile-time flags which can be added to `IP_Conf.h`. `IP_Conf.h` is the main configuration file for the TCP/IP stack.

14.1 Runtime configuration

Every driver folder includes a configuration file with implementations of runtime configuration functions explained in this chapter. These functions can be customized.

14.1.1 IP_X_Configure()

Description

Helper function to prepare and configure the TCP/IP stack.

Prototype

```
void IP_X_Config (void);
```

Additional information

This function is called by the startup code of the TCP/IP stack from `IP_Init()`. Refer to *IP_Init()* on page 79 for more information.

Example

```
/*********************************************************************
*
*       IP_X_Config
*
*  Function description
*    This function is called by the IP stack during IP_Init().
*
*  Typical memory/buffer configurations:
*    Microcontroller system, size optimized
*      #define ALLOC_SIZE 0x3000                // 12 KBytes RAM
*      mtu = 576;                               // 576 is minimum acc.
*                                               // to RFC, 1500 is max. for Ethernet
*      IP_SetMTU(0, mtu);                       // Maximum Transmission Unit is 1500
*                                               // for ethernet by default
*      IP_AddBuffers(8, 256);                   // Small buffers.
*      IP_AddBuffers(4, mtu + 16);              // Big buffers. Size should be mtu
*                                               // + 16 byte for ethernet header
*                                               // (2 bytes type, 2*6 bytes MAC,
*                                               // 2 bytes padding)
*      IP_ConfTCPSpace(1 * 1024, 1 * 1024);     // Define TCP Tx and Rx window size
*
*    Microcontroller system, speed optimized or multiple connections
*      #define ALLOC_SIZE 0x6000                // 24 KBytes RAM
*      mtu = 1500;                              // 576 is minimum acc. to RFC,
*                                               // 500 is max. for Ethernet
*      IP_SetMTU(0, mtu);                       // Maximum Transmission Unit is 1500
*                                               // for ethernet by default
*      IP_AddBuffers(12, 256);                  // Small buffers.
*      IP_AddBuffers(6, mtu + 16);              // Big buffers. Size should be mtu
*                                               // + 16 byte for ethernet header
*                                               // (2 bytes type, 2*6 bytes MAC,
*                                               // 2 bytes padding)
*      IP_ConfTCPSpace(4 * 1024, 4 * 1024);     // Define TCP Tx and Rx window size

*  System with lots of RAM
*      #define ALLOC_SIZE 0x20000               // 128 KBytes RAM
*      mtu = 1500;                              // 576 is minimum acc. to RFC,
*                                               // 1500 is max. for Ethernet
*      IP_SetMTU(0, mtu);                       // Maximum Transmission Unit is 1500
*                                               // for ethernet by default
*      IP_AddBuffers(50, 256);                  // Small buffers.
*      IP_AddBuffers(50, mtu + 16);             // Big buffers. Size should be mtu
*                                               // + 16 byte for ethernet header
*                                               // (2 bytes type, 2*6 bytes MAC,
*                                               // 2 bytes padding)
*      IP_ConfTCPSpace(8 * 1024, 8 * 1024);     // Define TCP Tx and Rx window size
*/
void IP_X_Config(void) {
  IP_AssignMemory(_aPool, sizeof(_aPool));    // Assigning memory
  IP_AddEtherInterface(&IP_Driver_STR912);    // Add ethernet driver
  IP_SetHWAddr("\x00\x22\x33\x44\x55\x66");   // MAC addr: Needs to be unique
                                              // for production units
  //
  // Use DHCP client or define IP address, subnet mask,
  // gateway address and DNS server according to the
```



```
  // requirements of your application.
  //
  IP_DHCPC_Activate(0, "TARGET", NULL, NULL);
//  IP_SetAddrMask(0xC0A805E6, 0xFFFF0000);     // Assign IP addr. and subnet mask
//  IP_SetGWAddr(0, 0xC0A80201);                // Set gateway address
//  IP_DNS_SetServer(0xCC98B84C);               // Set DNS server address,
                                                // for example 204.152.184.76
  //
  // Add protocols to the stack
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
  //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
  IP_AddBuffers(20, 256);                     // Small buffers.
  IP_AddBuffers(8, 1536);                     // Big buffers. Size should be 1536 to
                                              // allow a full ether packet to fit.
  IP_ConfTCPSpace(6 * 1024, 4 * 1024);        // Define the TCP Tx and Rx window size
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  //
  IP_SetWarnFilter(0xFFFFFFFF);               // 0xFFFFFFFF: Output all warnings.
  IP_SetLogFilter(IP_MTYPE_INIT               // Output all messages from init
                | IP_MTYPE_LINK_CHANGE        // Output a msg if link status changes
                | IP_MTYPE_DHCP               // Output general DHCP status messages
                );
}
```

14.1.2 Driver handling

IP_X_Config() is called at initialization of the TCP/IP stack. It is called by the IP stack during IP_Init(). IP_X_Config() should help to bundle the process of adding and configuring the driver.

14.1.3 Memory and buffer assignment

The total memory requirements of the TCP/IP stack can basically be computed as the sum of the following components:

Description	ROM
IP-Stack core	app. 200 bytes
Sockets	n * app. 200 bytes
UDP connection	n * app. 100 bytes
TCP/ connection	n * app. 200 bytes + RAM for TCP Window

14.1.3.1 RAM for TCP window

The data for the TCP window is typically stored in large buffers. The number of large buffers required is typically:

RxWindowSize / BigBufferSize

This amount of buffers (and RAM for these buffers) is needed for every simultaneously active TCP connection, where "active" means sending & receiving data.

14.1.3.2 Required buffers

Most of the RAM used by the stack is used for packet buffers. Packet buffers are used to hold incoming and outgoing packets and data in receive and transmit windows of TCP connections.

Example configuration - Extremly small (4 Kbytes)

This configuration is the smallest available or at least very close. It is intended to be used on MCUs with very little RAM and can be used for applications which are designed for a very low amount of traffic.

```
#define ALLOC_SIZE 0x1000                       // 4 Kbytes RAM
mtu = 576;                                      // 576 is minimum acc.
                                                // to RFC, 1500 is max. for Ethernet
IP_SetMTU(0, mtu);                              // Maximum Transmission Unit is 1500
                                                // for ethernet by default
IP_AddBuffers(4, 256);                          // Small buffers.
IP_AddBuffers(2, mtu + 16);                     // Big buffers. Size should be mtu
                                                // + 16 byte for ethernet header
                                                // (2 bytes type, 2*6 bytes MAC,
                                                // 2 bytes padding)
IP_ConfTCPSpace(1 * (mtu-40), 1 * (mtu-40));    // Define TCP Tx and Rx window size
```

Example configuration - Small (12 Kbytes)

This configuration is a small configuration intended to be used on MCUs with little RAM and can be used for applications which are designed for a medium amount of traffic.

```
#define ALLOC_SIZE 0x3000                       // 12 Kbytes RAM
mtu = 576;                                      // 576 is minimum acc.
                                                // to RFC, 1500 is max. for Ethernet
IP_SetMTU(0, mtu);                              // Maximum Transmission Unit is 1500
                                                // for ethernet by default
IP_AddBuffers(8, 256);                          // Small buffers.
IP_AddBuffers(4, mtu + 16);                     // Big buffers. Size should be mtu
                                                // + 16 byte for ethernet header
                                                // (2 bytes type, 2*6 bytes MAC,
                                                // 2 bytes padding)
IP_ConfTCPSpace(2 * (mtu-40), 2 * (mtu-40));    // Define TCP Tx and Rx window size
```

Example configuration - Normal (24 Kbytes)

This configuration is a typical configuration for many MCUs that have a fair amount of internal RAM. It can be used for applications which are designed for a higher amount of traffic and/or multiple client connections.

```
#define ALLOC_SIZE 0x6000                       // 24 Kbytes RAM
mtu = 1500;                                     // 576 is minimum acc. to RFC,
                                                // 500 is max. for Ethernet
IP_SetMTU(0, mtu);                              // Maximum Transmission Unit is 1500
                                                // for ethernet by default
IP_AddBuffers(12, 256);                         // Small buffers.
IP_AddBuffers(6, mtu + 16);                     // Big buffers. Size should be mtu
                                                // + 16 byte for ethernet header
                                                // (2 bytes type, 2*6 bytes MAC,
                                                // 2 bytes padding)
IP_ConfTCPSpace(3 * (mtu-40), 3 * (mtu-40));    // Define TCP Tx and Rx window size
```

Example configuration - Large (128 Kbytes)

This configuration is a large configuration intended to be used on MCUs with many external RAM. It can be used for applications which are designed for a high amount of traffic and multiple client/server connections at the same time.

```
#define ALLOC_SIZE 0x20000                      // 128 Kbytes RAM
mtu = 1500;                                     // 576 is minimum acc. to RFC,
                                                // 1500 is max. for Ethernet
IP_SetMTU(0, mtu);                              // Maximum Transmission Unit is 1500
                                                // for ethernet by default
IP_AddBuffers(50, 256);                         // Small buffers.
IP_AddBuffers(50, mtu + 16);                    // Big buffers. Size should be mtu
                                                // + 16 byte for ethernet header
                                                // (2 bytes type, 2*6 bytes MAC,
                                                // 2 bytes padding)
IP_ConfTCPSpace(6 * (mtu-40), 6 * (mtu-40));    // Define TCP Tx and Rx window size
```


14.2 Compile-time configuration

The following types of configuration macros exist:

Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

Numerical values "N"

Numerical values are used somewhere in the code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

Function replacements "F"

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).

14.2.1 Compile-time configuration switches

Type	Symbolic name	Default	Description
System configuration macros			
N	IP_IS_BIGENDIAN	--	Macro to define if a big endian target is used.
Debug macros			
N	IP_DEBUG	0	Macro to define the debug level of the embOS/IP build. Refer to *Debug level* on page 260 for a description of the different debug level.
Optimization macros			
F	IP_CKSUM	IP_cksum (C- routine in IP stack)	Macro to define an optimized checksum routine to speed up the stack. An optimized checksum routine is typically implemented in assembly language. Optimized versions for the GNU, IAR and ADS compilers are supplied.
F	IP_MEMCPY	memcpy (C-routine in standard C-library)	Macro to define an optimized memcpy routine to speed up the stack. An optimized memcpy routine is typically implemented in assembly language. Optimized version for the IAR compiler is supplied.
F	IP_MEMSET	memset (C-routine in standard C-library)	Macro to define an optimized memset routine to speed up the stack. An optimized memset routine is typically implemented in assembly language.

Type	Symbolic name	Default	Description
F	IP_MEMMOVE	memmove (C-routine in standard C-library)	Macro to define an optimized memmove routine to speed up the stack. An optimized memmove routine is typically implemented in assembly language.
F	IP_MEMCMP	memcmp (C-routine in standard C-library)	Macro to define an optimized memcmp routine to speed up the stack. An optimized memcmp routine is typically implemented in assembly language.

14.2.2 Debug level

embOS/IP can be configured to display debug information at higher debug levels to locate a problem (Error) or potential problem. To display information, embOS/IP uses the logging routines (see chapter *Debugging* on page 433). These routines can be blank, they are not required for the functionality of embOS/IP. In a target system, they are typically not required in a release (production) build, since a production build typically uses a lower debug level.

If `(IP_DEBUG == 0)`: used for release builds. Includes no debug options.
If `(IP_DEBUG == 1)`: `IP_PANIC() is mapped to IP_Panic()`.
If `(IP_DEBUG >= 2)`: `IP_PANIC() is mapped to IP_Panic()` and logging support is activated.

Chapter 15

Web server (Add-on)

The embOS/IP web server is an optional extension to embOS/IP. The web server can be used with embOS/IP or with a different TCP/IP stack. All functions that are required to add a web server task to your application are described in this chapter.

15.1 embOS/IP web server

The embOS/IP web server is an optional extension which adds the HTTP protocol to the stack. It combines a maximum of performance with a small memory footprint. The web server allows an embedded system to present web pages with dynamically generated content. It comes with all features typically required by embedded systems: multiple connections, authentication, forms and low RAM usage. RAM usage has been kept to a minimum by smart buffer handling.

The web server implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 1945]	HTTP - Hypertext Transfer Protocol -- HTTP/1.0 Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1945.txt*
[RFC 2616]	HTTP - Hypertext Transfer Protocol -- HTTP/1.1 Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt*

The following table shows the contents of the embOS/IP web server root directory:

Directory	Content
`Application\`	Contains the example application to run the web server with embOS/IP.
`Config`	Contains the web server configuration file. Refer to *Configuration* on page 283 for detailed information.
`Inc`	Contains the required include files.
`IP`	Contains the web server sources, `IP_Webserver.c`, `IP_Webserver.h and IP_UTIL_BASE64.c`, `IP_UTIL.h`.
`IP\FS\`	Contains the sources for the file system abstraction layer and the read-only file system. Refer to *File system abstraction layer* on page 456 for detailed information.
`Windows\Webserver\`	Contains the source, the project files and an executable to run embOS/IP web server on a Microsoft Windows host. Refer to *Using the web server sample* on page 267 for detailed information.

Supplied directory structure of embOS/IP web server package

15.2 Feature list

- Low memory footprint.
- Dynamic web pages (Server Side Includes).
- Authentication supported.
- Forms: POST and GET support.
- Multiple connections supported.
- r/o file system included.
- HTML to C converter included.
- Independent of the file system: any file system can be used.
- Independent of the TCP/IP stack: any stack with sockets can be used.
- Demo with authentication, various forms, dynamic pages included.
- Project for executable on PC for Microsoft Visual Studio included.

15.3 Requirements

TCP/IP stack

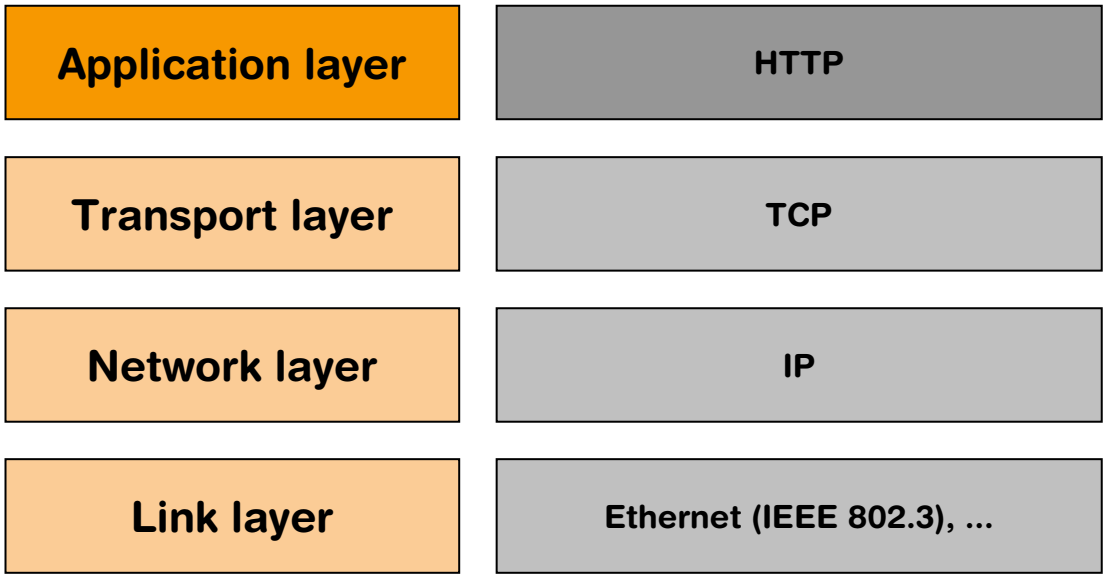
The embOS/IP web server requires a TCP/IP stack. It is optimized for embOS/IP, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of embOS/IP.

Multi tasking

The web server needs to run as a separate thread. Therefore, a multi tasking system is required to use the embOS/IP web server.

15.4 HTTP backgrounds

It is a communication protocol originally designed to transfer information via hypertext pages. The development of HTTP is coordinated by the IETF (Internet Engineering Task Force) and the W3C (World Wide Web Consortium). The current protocol version is 1.1.



15.4.1 HTTP communication basics

HTTP is a challenge and response protocol. A client initiates a TCP connection to the web server and sends a HTTP request. A HTTP request starts with a method token. *[RFC 2616]* defines 8 method tokens. The method token indicates the method to be performed on the requested resource. embOS/IP web server supports all methods which are typically required by an embedded web server.

HTTP method	Description
GET	The GET method means that it retrieves whatever information is identified by the Request-URI.
HEAD	The HEAD method means that it retrieves the header of the content which is identified by the Request-URI.
POST	The POST method submits data to be processed to the identified resource. The data is included in the body of the request.

Table 15.1: Supported HTTP methods

The following example shows parts of a HTTP session, where a client (for example, `192.168.1.75`) asks the embOS/IP web server for the hypertext page `example.html`. The request is followed by a blank line, so that the request ends with a double newline, each in the form of a carriage return followed by a line feed.

```
GET /example.html HTTP/1.1
Host: 192.168.1.75
```

The first line of every response message is the Status-Line, consisting of the protocol version followed by a numeric status code. The Status-Line is followed by the content-type, the server, expiration and the transfer-encoding. The server response ends with an empty line, followed by length of content that should be transferred. The length indicates the length of the web page in bytes.


```
HTTP/1.1 200 OK
Content-Type: text/html
Server: embOS/IP
Expires: THU, 26 OCT 1995 00:00:00 GMT
Transfer-Encoding: chunked

A3
```

Thereafter, the web server sends the requested hypertext page to the client. The zero at the end of the web page followed by an empty line signalizes that the transmission of the requested web page is complete.

```
<HTML>
  <HEAD>
    <TITLE>embOS/IP examples</TITLE>
  </HEAD>
  <BODY>
    <CENTER>
      <H1>Website: example.htm</H1>
    </CENTER>
  </BODY>
</HTML>
0
```

15.4.2 HTTP status codes

The first line of a HTTP response is the Status-Line. It consists of the used protocol version, a status code and a short textual description of the Status-Code. The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- 1xx: Informational - Request received, continuing process.
- 2xx: Success - The action was successfully received, understood, and accepted.
- 3xx: Redirection - Further action must be taken in order to complete the request.
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled.
- 5xx: Server Error - The server failed to fulfill an apparently valid request.

Refer to *[RFC 2616]* for a complete list of defined status-codes. embOS/IP web server supports a subset of the defined HTTP status codes. The following status codes are implemented:

Status code	Description
200	OK. The request has succeeded.
401	Unauthorized. The request requires user authentication.
404	Not found. The server has not found anything matching the Request-URI.
501	Not implemented. The server does not support the HTTP method.
503	Service unavailable. The server is currently unable to handle the request due to a temporary overloading of the server.

Table 15.2: embOS/IP status codes

15.5 Using the web server sample

Ready to use examples for Microsoft Windows and embOS/IP are supplied. If you use another TCP/IP stack, the sample `OS_IP_Webserver.c` has to be adapted.

The web server itself does not handle multiple connections. This is part of the application and is included in the `OS_IP_Webserver.c` sample.

The sample application opens a port which listens on port 80 until an incoming connection is detected in a parent task that accepts new connections (or rejects them if no more connections can be accepted).

For each accepted client connection, the parent task creates a child task running `IP_WEBS_Process()` in a seperated context that will then process the request of the connected client (for example a browser). This way the parent task is ready to handle further incoming connections on port 80.

Therefore the sample uses n client connections + one for the parent task.

Some browsers may open multiple connections and do not even intend to close the connection. They rather keep the connections open for further data that might be requested. To give other clients a chance, a special handling is implemented in the web server.

The embOS/IP web server has two functions for processing a connection in a child task:

- `IP_WEBS_Process()`, that allows a connection to stay open even after all data has been sent from the target. The connection will stay open as long as the client does not close it.
- `IP_WEBS_ProcessLast()`, that will close the connection once the target has sent all data requested. This is used by the web server sample for the last free connection available. This ensures that at least one connection will be available after it has been served to accept further clients.

In addition to available connections that can be served directly, a feature called "backlogging" can be used.
This means additional connections will be accepted (SYN/ACK is sent from target) but not yet processed. They will be processed as soon as a free connection becomes available once a child task has served the clients request and has been closed.
Connections in backlog will be kept active until the client side sends a reset due to a possible timeout in the client.

The example application uses a read-only file system to make web pages available. Refer to *File system abstraction layer* on page 456 and *File system abstraction layer* on page 456 for detailed information about the read-only file system.

15.5.1 Using the Windows sample

If you have MS Visual C++ 6.00 or any later version available, you will be able to work with a Windows sample project using embOS/IP web server. If you do not have the Microsoft compiler, an precompiled executable of the web server is also supplied.

Building the sample program

Open the workspace `Start_Webserver.dsw` with MS Visual Studio (for example, double-clicking it). There is no further configuration necessary. You should be able to build the application without any error or warning message.

The server uses the IP address of the host PC on which it runs. Open a web browser and connect by entering the IP address of the host (`127.0.0.1`) to connect to the web server.

15.5.2 Running the web server example on target hardware

The embOS/IP web server sample application should always be the first step to check the proper function of the web server with your target hardware.

Add all source files located in the following directories (and their subdirectories) to your project and update the include path:

- `Application`
- `Config`
- `Inc`
- `IP`
- `IP\IP_FS\FS_RO\`
- `IP\IP_FS\FS_RO\Generated\`

It is recommended that you keep the provided folder structure.

The sample application can be used on the most targets without the need for changing any of the configuration flags. The server processes up to three connections using the default configuration.

Note: Three connections mean that the target can handle up to three targets in parallel, if every target uses only one connection. Because a single web browser often attempts to open more then one connection to a web server to request the files (.gif, .jpeg, etc.) which are included in the requested web page, the number of possible parallel connected targets is less than the number of possible connections.

Every connection is handled in an separate task. Therefore, the web server uses up to four tasks in the default configuration, one task which listens on port 80 and accepts connections and three tasks to process the accepted connections. To modify the number of connections, only the macro `MAX_CONNECTIONS` has to be modified.

The supplied sample web pages `index.htm`, `embos.htm` and `stats.htm` include dynamic content, refer to *Common Gateway Interface (CGI)* on page 270 for detailed information about the implementation of dynamic content.

15.5.3 Changing the file system type

By default, the web server uses the supplied read-only file system. If a real file system like emFile should be used to store the web pages, you have to modify the function `_WebServerChildTask()` of the example `OS_IP_Webserver.c`.

```
/*********************************************************************
*
*       _WebServerChildTask
*
*/
static void _WebServerChildTask(void * Context) {
  long Sock;
  int Opt;

  _pFS_API = &IP_FS_ReadOnly;
  Sock = (long)Context;
  Opt = 1;
  setsockopt(Sock, SOL_SOCKET, SO_KEEPALIVE, &Opt, sizeof(Opt));
  if (_ConnectCnt < MAX_CONNECTIONS) {
    IP_WEBS_Process(_Send, _Recv, Context, _pFS_API, &_Application);
  } else {
    IP_WEBS_ProcessLast(_Send, _Recv, Context, _pFS_API, &_Application);
  }
  _closesocket(Sock);
  _AddToConnectCnt(-1);
  OS_Terminate(0);
}
```

The usage of the read-only file system is configured with the following line:

```
_pFS_API = &IP_FS_ReadOnly;
```

To use emFile as file system for your web server application, add the emFile abstraction layer `IP_FS_FS.c` to your project and change the line to:

```
_pFS_API = &IP_FS_FS;
```

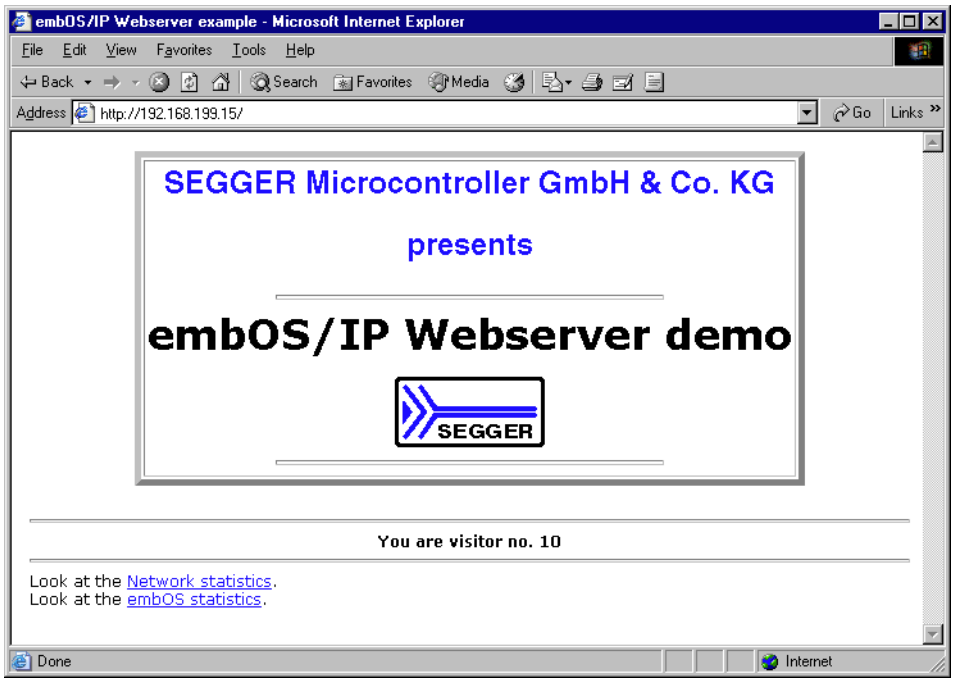
Refer to *File system abstraction layer* on page 456 and *File system abstraction layer* on page 456 for detailed information about the emFile and read-only file system abstraction layer.

15.6 Dynamic content

embos/IP supports two different approaches to implement dynamic content in your web server application. A Common Gateway Interface (CGI) like interface for static HTML pages with dynamic elements and virtual files which are completely generated from the application.

15.6.1 Common Gateway Interface (CGI)

A Common Gateway Interface (CGI) like interface is used to implement dynamic content in web pages. Every web page will be parsed by the server each time a request is received. The server searches the web page for a special tag. In the default configuration, the searched tag starts `<!--#exec cgi="` and ends with `"-->`. The tag will be analyzed and the parameter will be extracted. This parameter specifies a server-side command and will be given to the user application, which can handle the command. The following screenshot shows the example page `index.htm`.



The HTML source for the page includes the following line:

```
<!--#exec cgi="Counter"-->
```

When the web page is requested, the server parses the tag and the parameter `Counter` is searched for in an array of structures of type `WEBS_CGI`. The structure includes a string to identify the command and a pointer to the function which should be called if the parameter is found.

```
typedef struct {
  const char * sName;   // e.g. "Counter"
  void  (*pf)(WEBS_OUTPUT * pOutput, const char * sParameters, const char * sValue);
} WEBS_CGI;
```

In the example, `Counter` is a valid parameter and the function `_callback_ExecCounter` will be called. You need to implement `the WEBS_CGI` array and the callback functions in your application.

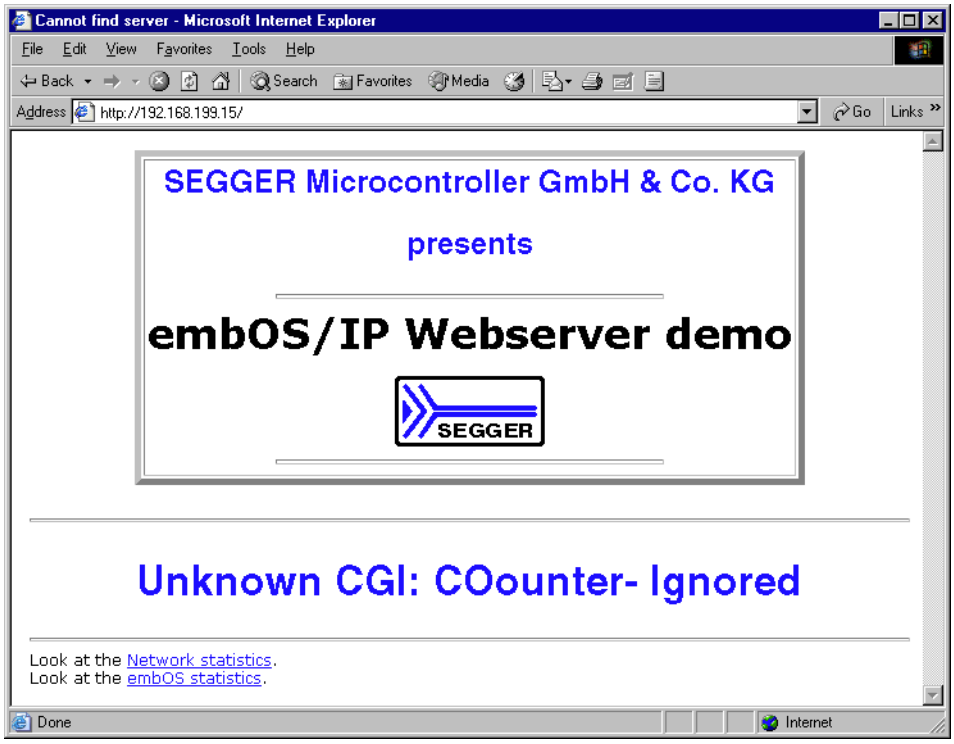
```
static const WEBS_CGI _aCGI[] = {
  {"Counter"    , _callback_ExecCounter  },
  {"GetOSInfo" , _callback_ExecGetOSInfo},
  {"GetIPAddr" , _callback_ExecGetIPAddr},
  {NULL}
};
```


`ExecCounter()` is a simple example of how to use the CGI feature. It returns a string that includes the value of a variable which is incremented with every call to `ExecCounter()`.

```
void ExecCounter(       WEBS_OUTPUT * pOutput,
                  const char        * sParameters,
                  const char        * sValue ) {
  char ac[40];
  static char Cnt = 1;

  sprintf(ac, "You are visitor no.: %d", Cnt);
  IP_WEBS_SendString(pOutput, ac);
  Cnt++;
}
```

If the web page includes the CGI tag followed by an unknown command (for example, a typo like `COounter` instead of `Counter` in the source code of the web page) an error message will be sent to the client.



15.6.1.1 Add new CGI functions to your web server application

To define new CGI functions, three things have to be done.

1. Add a new command name which should be used as tag to the `WEBS_CGI` structure. For example: `UserCGI`

   ```
   static const WEBS_CGI _aCGI[] = {
     {"Counter"   , _callback_ExecCounter  },
     {"GetOSInfo" , _callback_ExecGetOSInfo},
     {"GetIPAddr" , _callback_ExecGetIPAddr},
     {"UserCGI"   , _callback_ExecUserCGI  },
     {NULL}
   };
   ```

2. Implement the new function in your application source code.

   ```
   void _callback_ExecUserCGI(       WEBS_OUTPUT * pOutput,
                              const char        * sParameters
                              const char        * sValue ) {
     /* Add application code here */
   }
   ```

3. Add the new tag to the source code of your web page:

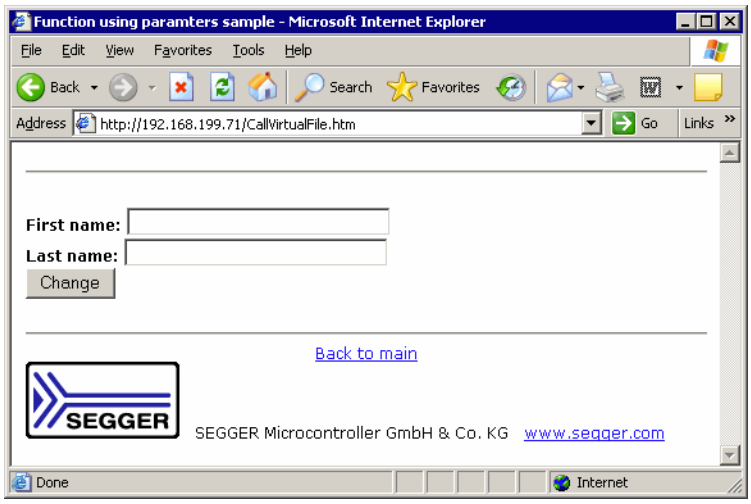
   ```
   <!--#exec cgi="UserCGI"-->
   ```


15.6.2 Virtual files

embOS/IP supports virtual files. A virtual file is not a real file which is stored in the used file system. It is a function which is called instead. The function generates the content of a file and sends it to the client.

The web server checks the extension of all requested files, the extension `.cgi` is by default used for virtual files. To change the extension that is used to detect a virtual file, refer to *IP_WEBS_SetFileInfoCallback()* on page 294 for detailed information.

The embOS/IP web server comes with an example (`CallVirtualFile.htm`) that requests a virtual file. The sample web page contains a form with two input test fields, named `FirstName` and `LastName`, and a button to transmit the data to the server.



When the button on the web page is pressed, the file Send.cgi is requested. The embOS/IP Web server recognizes the extension `.cgi`, checks if a virtual file with the name `Send.cgi` is defined and calls the defined function. The function in the example is `_callback_SendCGI` and gets the string `FirstName=Foo&LastName=Bar` as parameter.

```
typedef struct {
  const char * sName;
  void  (*pf)(WEBS_OUTPUT * pOutput, const char * sParameters);
} WEBS_VFILES;
```

In the example, `Send.cgi` is a valid URI and the function `_callback_SendCGI` will be called.

```
static const WEBS_VFILES _aVFiles[]  = {
  {"Send.cgi", _callback_SendCGI  },
  NULL
};
```

The virtual file `Send.cgi` gets two parameters. The strings entered in the input fields `Firstname` and `LastName` are transmitted with the URI. For example, you enter `Foo` in the first name field and `Bar` for last name and push the button. The browser will transmit the following string to our web server:

```
Send.cgi?FirstName=Foo&LastName=Bar
```

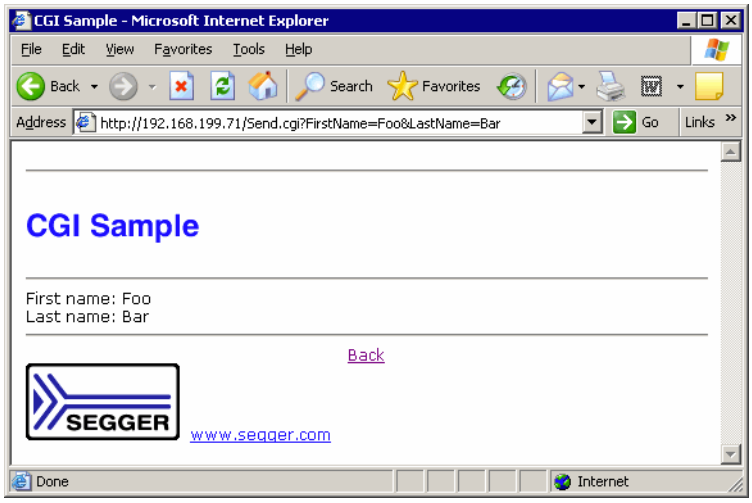
You can parse the string and use it in the way you want to. In the example we parse the string and output the values on a web page which is build from the function `_callback_SendCGI()`.

```
static void _callback_SendCGI(WEBS_OUTPUT * pOutput, const char * sParameters) {
  char aPara0[32];
  char aValue0[32];
  char aPara1[32];
  char aValue1[32];
  int r;
```



```
  IP_WEBS_SendString(pOutput, "<HTML><HEAD><TITLE>CGI Sample</TITLE></HEAD>");
  IP_WEBS_SendString(pOutput, "<style type=\"text/css\">
                               H1, H2, H3, H4 { color: blue }
                               H1, H2, H3, H4, H5 {font-family: Helvetica;}
                               PRE {color: black;  margin-left: 2%; font-size=150%}
                               BODY {color: black;  margin-left: 2%; }
                               </style>");
  IP_WEBS_SendString(pOutput, "<HR><H2>CGI Sample</H2><HR><BODY>First name: ");
  r = IP_WEBS_GetParaValue(sParameters, 0, aPara0,
                           sizeof(aPara0), aValue0, sizeof(aValue0));
  if (r == 0) {
    IP_WEBS_SendString(pOutput, aValue0);
  }
  IP_WEBS_SendString(pOutput, "<BR>Last name: ");
  r =  IP_WEBS_GetParaValue(sParameters, 1, aPara1,
                            sizeof(aPara1), aValue1, sizeof(aValue1));
  if (r == 0) {
    IP_WEBS_SendString(pOutput, aValue1);
  }
  IP_WEBS_SendString(pOutput, "<BR>");
  IP_WEBS_SendString(pOutput, "<HR><CENTER>
                               <A HREF=\"CallVirtualFile.htm\">Back</A>
                               </CENTER><IMG SRC=\"segger.gif\">  
                               <A HREF=\"http://www.segger.com\">www.segger.com</A>
                               </BODY></HTML>");
}
```

The output of `_callback_SendCGI()` should be similar to:



15.7 Authentication

"HTTP/1.0", includes the specification for a Basic Access Authentication scheme. The basic authentication scheme is a non-secure method of filtering unauthorized access to resources on an HTTP server, because the user name and password are passed over the network as clear text. It is based on the assumption that the connection between the client and the server can be regarded as a trusted carrier. As this is not generally true on an open network, the basic authentication scheme should be used accordingly.

The basic access authentication scheme is described in:

RFC#	Description
[RFC 2617]	HTTP Authentication: Basic and Digest Access Authentication Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2617.txt*

The "basic" authentication scheme is based on the model that the client must authenticate itself with a user-ID and a password for each realm. The realm value should be considered an opaque string which can only be compared for equality with other realms on that server. The server will service the request only if it can validate the user-ID and password for the protection space of the Request-URI. There are no optional authentication parameters.

Upon receipt of an unauthorized request for a URI within the protection space, the server should respond with a challenge like the following:

```
WWW-Authenticate: Basic realm="Embedded web server"
```

where `"embOS/IP embedded web server"` is the string assigned by the server to identify the protection space of the Request-URI. To receive authorization, the client sends the user-ID and password, separated by a single colon (":") character, within a base64 encoded string in the credentials.

If the user agent wishes to send the user-ID "`user`" and password "`pass`", it would use the following header field:

```
Authorization: Basic dXNlcjpwYXNz
```


15.7.1 Authentication example

The client requests a resource for which authentication is required:

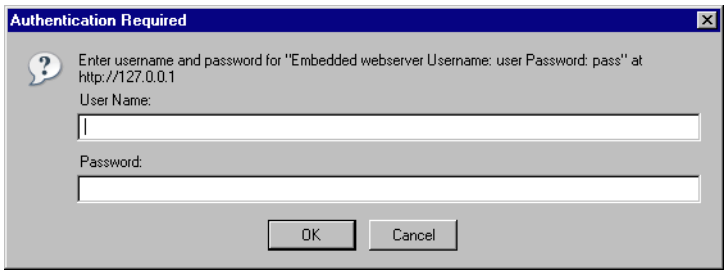
```
GET /conf/Authen.htm HTTP/1.1
Host: 192.168.1.75
```

The server answers the request with a "401 Unauthorized" status page. The header of the 401 error page includes an additional line WWW-Authenticate. It includes the realm for which the proper user name and password should be transmitted from the client (for example, a web browser).

```
HTTP/1.1 401 Unauthorized
Date: Mon, 04 Feb 2008 17:00:44 GMT
Server: embOS/IP
Accept-Ranges: bytes
Content-Length: 695
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug
WWW-Authenticate: Basic realm="embOS/IP embedded web server"

<HTML>
<HEAD><TITLE>401 Unauthorized</TITLE></HEAD>
<BODY>
<H1>401 Unauthorized</H1>
Browser not authentication-capable or authentication failed.<P>
</BODY>
</HTML>
```

The client interprets the header and opens a dialog box to enter the user name and password combination for the realm of the resource.



Note: The embOS/IP web server example always uses the following user name and the password combination: `User Name: user - Password: pass`

Enter the proper user name/password combination for the requested realm and confirm with the **OK** button. The client encodes the user name/password combination to a base64 encoded string and requests the resource again. The request header is enhanced by the following line: `Authorization: Basic dXNlcjpwYXNz`

```
GET /conf/Authen.htm HTTP/1.1
Host: 192.168.1.75
Authorization: Basic dXNlcjpwYXNz
```

The server decodes the user name/password combination and checks if the decoded string matches to the defined user name/password combination of the realm. If the strings are identical, the server delivers the page. If the strings are not identical, the server answers again with a "401 Unauthorized" status page.


```
HTTP/1.1 200 OK
Content-Type: text/html
Server: embOS/IP
Expires: THU, 26 OCT 1995 00:00:00 GMT
Transfer-Encoding: chunked

200
<HTML>
  <HEAD>
    <TITLE>web server configuration</TITLE>
  </HEAD>
  <BODY>
    <!-- Content of the page -->
  </BODY>
</HTML>

0
```

15.7.2 Configuration of the authentication

The embOS/IP web server checks the access rights of every resource before returning it. The user can define different realms to separate different parts of the web server resources. An array of `WEBS_ACCESS_CONTROL` structures has to be implemented in the user application. Refer to *Structure WEBS_ACCESS_CONTROL* on page 312 for detailed information about the elements of the `WEBS_ACCESS_CONTROL` structure. If no authentication should be used, the array includes only one entry for the root path.

```
WEBS_ACCESS_CONTROL _aAccessControl[] = {
  { "/", NULL, NULL },
  0
};
```

To define a realm "`conf`", an additional `WEBS_ACCESS CONTROL` entry has to be implemented.

```
WEBS_ACCESS_CONTROL _aAccessControl[] = {
  { "/conf/", "Login for configuration", "user:pass" },
  { "/", NULL, NULL },
  0
};
```

The string `"Login for configuration"` defines the realm. `"user:pass"` is the user name/password combination stored in one string.

15.8 Form handling

The embOS/IP web server supports both `POST` and `GET` actions to receive form data from a client. `POST` submits data to be processed to the identified resource. The data is included in the body of the request. `GET` is normally only used to requests a resource, but it is also possible to use `GET` for actions in web applications. Data processing on server side might create a new resource or update existing resources or both.

Every HTML form consists of input items like textfields, buttons, checkboxes, etc. Each of these input items has a `name` tag. When the user places data in these items in the form, that information is encoded into the form data. Form data is a stream of `<name>=<value>` pairs separated by the "`&`" character. The value each of the input item is given by the user is called the value. The `<name>=<value>` pairs are URL encoded, which means that spaces are changed into "`+`" and special characters are encoded into hexadecimal values. Refer to *[RFC 1738]* for detailed information about URL encoding. The parsing and decoding of form data is handled by the embOS/IP web server. Thereafter, the server calls a callback function with the decoded and parsed strings as parameters. The responsibility to implement the callback function is on the user side.

Valid characters for CGI function names:

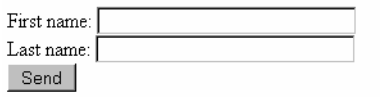
- A-Z
- a-z
- 0-9
- . _ -

Valid characters for CGI parameter values:

- A-Z
- a-z
- 0-9
- All URL encoded characters
- . _ - *()!$\

15.8.1 Simple form processing sample

The following example shows the handling of the output of HTML forms with your web server application. The example web page `ExampleGET.htm` implements a form with three inputs, two text fields and one button.



The HTML code of the web page as it is added to the server is listed below:

```
<html>
  <head><title>embOS/IP web server form example</title></head>
  <body>
    <form action="" method="GET">
      <p>
        First name:
        <input name="FirstName"
               type="text" size="30"
               maxlength="30"
               value="<!--#exec cgi="FirstName"-->"
        >
        <br>
        Last name:
        <input name="LastName"
               type="text"
               size="30"
               maxlength="30"
               value="<!--#exec cgi="LastName"-->"
        >
        <br>
        <input type="submit" value="Send">
      </p>
    </form>
  </body>
</html>
```

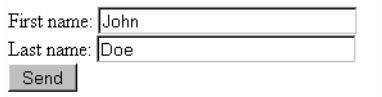
The action field of the form can specify a resource that the browser should reference when it sends back filled-in form data. If the action field defines no resource, the current resource will be requested again.

If you request the web page from the embOS/IP web server and check the source of the page in your web browser, the CGI parts "`<!--#exec cgi="FirstName"-->`" and "`<!--#exec cgi="LastName"-->`" will be executed before the page will be transmitted to the server, so that in the example the values of the `value=` fields will be empty strings.

The HTML code of the web page as seen by the web browser is listed below:

```
<html>
  <head><title>embOS/IP web server form example</title></head>
  <body>
    <form action="" method="GET">
      <p>
        First name:
        <input name="FirstName"
               type="text" size="30"
               maxlength="30"
               value=""
        >
        <br>
        Last name:
        <input name="LastName"
               type="text"
               size="30"
               maxlength="30"
               value=""
        >
        <br>
        <input type="submit" value="Send">
      </p>
    </form>
  </body>
</html>
```


To start form processing, you have to fill in the FirstName and the LastName field and click the Send button. In the example, the browser sends a GET request for the resource referenced in the form and appends the form data to the resource name as an URL encoded string. The form data is separated from the resource name by "?". Every <name>=<value> pair is separated by "&".



For example, if you type in the FirstName field John and Doe in the LastName field and confirm the input by clicking the Send button, the following string will be transmitted to the server and shown in the address bar of the browser.

```
http://192.168.1.230/ExampleGET.htm?FirstName=John&LastName=Doe
```

Note: If you use POST as HTTP method, the name>=<value> pairs are not shown in the address bar of the browser. The <name>=<value> pairs are in this case included in the entity body.

The embOS/IP web server parses the form data. The <name> field specifies the name of a CGI function which should be called to process the <value> field. The server checks therefore if an entry is available in the WEBS_CGI array.

```
static const WEBS_CGI _aCGI[] = {
  {"FirstName",   _callback_ExecFirstName},
  {"LastName",    _callback_ExecLastName },
  {NULL}
};
```

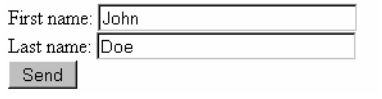
If an entry can be found, the specified callback function will be called.

The callback function for the parameter FirstName is defined as follow:

```
/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
static char _acFirstName[12];

/*********************************************************************
*
*       _callback_FirstName
*/
static void _callback_ExecFirstName(        WEBS_OUTPUT * pOutput,
                                    const char        * sParameters,
                                    const char        * sValue ) {
  if (sValue == NULL) {
    IP_WEBS_SendString(pOutput, _acFirstName);
  } else {
    _CopyString(_acFirstName, sValue, sizeof(_acFirstName));
  }
}
```

The function returns a string if sValue is NULL. If sValue is defined, it will be written into a character array. Because HTTP transmission methods GET and POST only transmit the value of filled input fields, the same function can be used to output a stored value of an input field or to set a new value. The example web page shows after entering and transmitting the input the new values of FirstName and LastName as value in the input fields.



The source of the web page as seen by the web browser is listed below:


```
<html>
  <head><title>embOS/IP web server form example</title></head>
  <body>
    <form action="" method="GET">
      <p>
        First name:
        <input name="FirstName"
               type="text" size="30"
               maxlength="30"
               value="John"
        >
        <br>
        Last name:
        <input name="LastName"
               type="text"
               size="30"
               maxlength="30"
               value="Doe"

        >
        <br>
        <input type="submit" value="Send">
      </p>
    </form>
  </body>
</html>
```

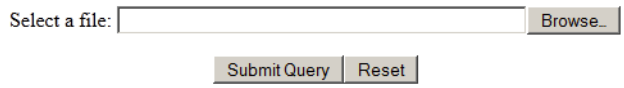

15.9 File upload

The embOS/IP web server supports file uploads from the client. For this to be possible a real file system has to be used and the define WEBS_SUPPORT_UPLOAD has to be defined to "1".

From the application side uploading a file in general is the same as for other form data as described in *Form handling* on page 277. For file uploading a <form> field with encoding of type multipart/form-data is needed. An upload form field may contain additional input fields that will be parsed just as if using a non upload formular and can be parsed in your callback using *IP_WEBS_GetParaValue()* on page 301 or by using *IP_WEBS_GetParaValuePtr()* on page 302.

15.9.1 Simple form upload sample

The following example shows the handling of file uploads with your web server application. The example web page Upload.htm implements a form with a file upload field.

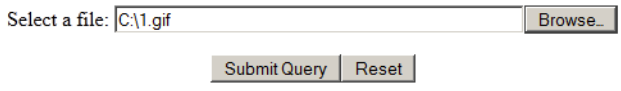


The HTML code of the web page as it is added to the server is listed below:

```
<HTML>
  <BODY>
    <CENTER>
      <P>
        <form action="Upload.cgi" method="post" enctype="multipart/form-data">
          <p>Select a file: <input name="Data" type="file">
          </p>
          <input type="submit"><input type="reset">
        </form>
      </P>
    </CENTER>
  </BODY>
</HTML>
```

The action field of the form can specify a resource that the browser should reference when it has finished handling the file upload. If the action field defines no resource, the current resource will be requested again.

To upload a file, you have to select a file by using the browse button and select a file to upload and click the Send button. In the example, the browser sends a POST request for the resource referenced in the form and appends the form and file data in an encoded string.



The embOS/IP web server parses additional form data passed besides the file to be uploaded. This works the same as handling form data described in *Form handling* on page 277. The action parameter of the <form> field specifies the name of a virtual file that should be processed. A callback can then be used to provide an answer page referring the state of the upload. The example below shows how to check the success of an upload using a virtual file prvided by the WEBS_VFILES array:

```
static const WEBS_VFILES _aVFiles[]  = {
 {"Upload.cgi", _callback_CGI_UploadFile  },
  { NULL, NULL }
};
```

If an entry can be found, the specified callback function will be called.

The callback function for the file Upload.cgi is defined as follow:

```
/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

/*********************************************************************
*
*       _callback_CGI_UploadFile
*/
static   void   _callback_CGI_UploadFile(WEBS_OUTPUT   *   pOutput,   const   char   *
sParameters) {
        int    r;
  const char * pFileName;
        int    FileNameLen;
   const  char  *  pState;          // Can be 0: Upload failed; 1: Upload succeeded;
Therefore we do not need to know the length, it will always be 1.

  IP_WEBS_SendString(pOutput, "<HTML><BODY>");
  r  = IP_WEBS_GetParaValuePtr(sParameters, 0, NULL, 0, &pFileName, &FileNameLen);
  r |= IP_WEBS_GetParaValuePtr(sParameters, 1, NULL, 0, &pState   , NULL);
  if (r == 0) {
    IP_WEBS_SendString(pOutput, "Upload of \"");
    IP_WEBS_SendMem(pOutput, pFileName, FileNameLen);
    if (*pState == '1') {
      IP_WEBS_SendString(pOutput, "\" successful!<br>");
      IP_WEBS_SendString(pOutput, "<a href=\"");
      IP_WEBS_SendMem(pOutput, pFileName, FileNameLen);
      IP_WEBS_SendString(pOutput, "\">Go to ");
      IP_WEBS_SendMem(pOutput, pFileName, FileNameLen);
      IP_WEBS_SendString(pOutput, "</a><br>");
    } else {
      IP_WEBS_SendString(pOutput, "\" not successful!<br>");
    }
  } else {
    IP_WEBS_SendString(pOutput, "Upload not successful!");
  }
 IP_WEBS_SendString(pOutput, "</BODY></HTML>");
}
```

In addition to the provided form fields from the upload form used two additional entries will be added to the end of the parameter list available for parsing:

- The original filename of the file uploaded
- The status of the upload process. This can be 0: Upload failed or 1: Upload succeeded.

The example web page shows after the upload has been finished.

Upload of "1.gif" successful!
Go to 1.gif

The source of the web page as seen by the web browser is listed below:

```
<HTML><BODY>
Upload of "1.gif" successful!<br>
<a href="1.gif">Go to 1.gif</a><br>
</BODY></HTML>
```


15.10 Configuration

The embOS/IP web server can be used without changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which match the requirements of most applications.

The following types of configuration macros exist:

Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

Numerical values "N"

Numerical values are used somewhere in the source code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

Alias "A"

A macro which operates like a simple text substitute. An example would be the define `U8`, which the preprocessor would replace with `unsigned char`.

Function replacements "F"

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the source code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).

15.10.1 Compile time configuration switches

Type	Symbolic name	Default	Description
F	WEBS_WARN	--	Defines a function to output warnings. In debug configurations (DEBUG == 1) WEBS_WARN maps to IP_Warnf_Application().
F	WEBS_LOG	--	Defines a function to output logging messages. In debug configurations (DEBUG == 1) WEBS_LOG maps to IP_Logf_Application().
N	WEBS_IN_BUFFER_SIZE	512	Defines the size of the input buffer. The input buffer is used to store the HTTP client requests.
N	WEBS_OUT_BUFFER_SIZE	512	Defines the size of the output buffer. The output buffer is used to store the HTTP response.
N	WEBS_PARA_BUFFER_SIZE	256	Defines the size of the buffer used to store the parameter/value string that is given to a virtual file. If virtual files are not used in your application, remove the definition from WEBS_Conf.h to save RAM.
N	WEBS_TEMP_BUFFER_SIZE	256	Defines the size of the TEMP buffer used internally by the web server.

Type	Symbolic name	Default	Description
N	WEBS_AUTH_BUFFER_SIZE	32	Defines the size of the buffer used to store the authentication string. Refer to *Authentication* on page 274 for detailed information about authentication.
N	WEBS_FILENAME_BUFFER_SIZE	32	Defines the size of the buffer used to store the filename strings.
B	WEBS_SUPPORT_UPLOAD	0/1	Defines if file upload is enabled. Defaults to 0: Not enabled, for source code shipments and 1: Enabled, for object shipments.

Status message web pages

The status message web pages are visualizations of the information transmitted to the client in the header of the web server response. Because these visualizations are not required for the functionality of the web server, the macros can be empty.

Type	Symbolic name	Default

A	WEBS_401_PAGE	`"<HTML>\n"  \`
		`"<HEAD>\n"  \`
		`"<TITLE>401 Unauthorized</TITLE>\n" \`
		`</HEAD>\n"  \`
		`"<BODY>\n"  \`
		`"<H1>401 Unauthorized</H1>\n" \`
		`"Browser not authentication-capable" \`
		`"or authentication failed.\n"  \`
		`"</BODY>\n" \`
		`"</HTML>\n"`

Type	Symbolic name	Default
A	WEBS_404_PAGE	"<HTML>\n" \ "<HEAD>\n" \ "<TITLE>404 Not Found</TITLE>\n" \ </HEAD>\n" \ "<BODY>\n" \ "<H1>404 Not Found</H1>\n" \ "The requested document was not " \ "found on this server.\n" \ "</BODY>\n" \ "</HTML>\n"
A	WEBS_501_PAGE	"<HTML>\n" \ "<HEAD>\n" \ "<TITLE>501 Not implemented</TITLE>\n" \ </HEAD>\n" \ "<BODY>\n" \ "<H1>Command is not implemented</H1>\n" \ "</BODY>\n" \ "</HTML>\n"
A	WEBS_503_PAGE	"<HTML>\n" \ "<HEAD>\n" \ "<TITLE>503 Connection limit reached</TITLE>\n" \ </HEAD>\n" \ "<BODY>\n" \ "<H1>503 Connection limit reached</H1>\n" \ "The max. number of simultaneous connections to " \ "this server reached.<P>\n" \ "Please try again later.\n" \ "</BODY>\n" \ "</HTML>\n"

15.11 API functions

Function	Description
IP_WEBS_Process()	Processes a HTTP request of a client.
IP_WEBS_ProcessLast()	Processes a HTTP request of a client and closes the connection thereafter.
IP_WEBS_OnConnectionLimit()	Outputs an error message to the connected client.
IP_WEBS_SendMem()	Sends data to a connected target.
IP_WEBS_SendString()	Sends a string to a connected target.
IP_WEBS_SendStringEnc()	Encodes and sends a string to a connected target.
IP_WEBS_SendUnsigned()	Sends an unsigned value to a connected target.
IP_WEBS_SetFileInfoCallback()	Sets a callback function to handle file information used by the web server.
IP_WEBS_AddFileTypeHook()	Adds a new file anme extension to MIME type correlation.
IP_WEBS_ConfigSendVFileHeader()	Configures automatic sending of a header based on the file name for virtual files.
IP_WEBS_ConfigSendVFileHookHeader()	Configures automatic sending of a header based on the file name for VFile hooks.
IP_WEBS_SendHeader()	Sends a header with data provided.
CGI/virtual file related functions	
IP_WEBS_CompareFileNameExt()	Checks the file name extension.
IP_WEBS_GetNumParas()	Returns the number of parameter/value pairs.
IP_WEBS_GetParaValue()	Gets a parameter value pair.
IP_WEBS_DecodeAndCopyStr()	Decodes an HTML encoded string and copy it into a buffer.
IP_WEBS_DecodeString()	Decodes an HTML encoded string.
IP_WEBS_AddVFileHook()	Adds a hook to serve a simple virtual file.
Utility functions	
IP_UTIL_BASE64_Decode()	Decodes a Base64 encoded string.
IP_UTIL_BASE64_Encode()	Encodes a string as a Base64 string.

Table 15.3: embOS/IP web server interface function overview

15.11.1 IP_WEBS_Process()

Description

Processes a HTTP request of a client.

Prototype

```
int IP_WEBS_Process (       IP_WEBS_tSend      pfSend,
                            IP_WEBS_tReceive   pfReceive,
                            void             * pConnectInfo,
                      const IP_WEBS_FS_API   * pFS_API
                      const WEBS_APPLICATION * pApplication);
```

Parameter

Parameter	Description
pfSend	[IN] Pointer to the function to be used by the server to send data to the client.
pfReceive	[IN] Pointer to the function to be used by the server to receive data from the client.
pConnectInfo	[IN] Pointer to the connection information.
pFS_API	[IN] Pointer to the used file system API.
pApplication	[IN] Pointer to a structure of type WEBS_APPLICATION.

Table 15.4: IP_WEBS_Process() parameter list

Return value

0 OK.

Additional Information

This function is part of the thread functionality of the web server. The following types are used as function pointers to the routines used to send and receive bytes from/to the client:

```
typedef int (*IP_WEBS_tSend)    (const unsigned char * pData,
                                                int    len,
                                                void * pConnectInfo);

typedef int (*IP_WEBS_tReceive) (const unsigned char * pData,
                                                int    len,
                                                void * pConnectInfo);
```

The send and receive functions should return the number of bytes successfully sent/ received to/from the client. The pointer pConnectInfo is passed to the send and receive routines. It can be used to pass a pointer to a structure containing connection information or to pass a socket number. For details about the parameter pFS_API and the IP_WEBS_FS_API structure, refer to *File system abstraction layer* on page 456. For details about the parameter pApplication and the WEBS_APPLICATION structure, refer to *Structure WEBS_APPLICATION* on page 313.

Refer to *IP_WEBS_ProcessLast()* on page 288 and *IP_WEBS_OnConnectionLimit()* on page 289 for further information.

15.11.2 IP_WEBS_ProcessLast()

Description

Processes a HTTP request of a client and closes the connection thereafter.

Prototype

```
int IP_WEBS_Process (       IP_WEBS_tSend      pfSend,
                            IP_WEBS_tReceive   pfReceive,
                            void             * pConnectInfo,
                      const IP_WEBS_FS_API   * pFS_API
                      const WEBS_APPLICATION * pApplication);
```

Parameter

Parameter	Description
pfSend	[IN] Pointer to the function to be used by the server to send data to the client.
pfReceive	[IN] Pointer to the function to be used by the server to receive data from the client.
pConnectInfo	[IN] Pointer to the connection information.
pFS_API	[IN] Pointer to the used file system API.
pApplication	[IN] Pointer to a structure of type WEBS_APPLICATION.

Table 15.5: IP_WEBS_Process() parameter list

Return value

0 OK.

Additional Information

This function is part of the thread functionality of the web server. This is typically called for the last available connection. In contrast to IP_WEBS_Process(), this function closes the connection as soon as the command is completed in order to not block the last connection longer than necessary and avoid connection-limit errors.

The following types are used as function pointers to the routines used to send and receive bytes from/to the client:

```
typedef int (*IP_WEBS_tSend)    (const unsigned char * pData,
                                                int    len,
                                                void * pConnectInfo);

typedef int (*IP_WEBS_tReceive) (const unsigned char * pData,
                                                int    len,
                                                void * pConnectInfo);
```

The send and receive functions should return the number of bytes successfully sent/received to/from the client. The pointer pConnectInfo is passed to the send and receive routines. It can be used to pass a pointer to a structure containing connection information or to pass a socket number. For details about the parameter pFS_API and the IP_WEBS_FS_API structure, refer to *File system abstraction layer* on page 456. For details about the parameter pApplication and the WEBS_APPLICATION structure, refer to *Structure WEBS_APPLICATION* on page 313.

Refer to *IP_WEBS_Process()* on page 287 and *IP_WEBS_OnConnectionLimit()* on page 289 for further information.

15.11.3 IP_WEBS_OnConnectionLimit()

Description

Outputs an error message to the connected client.

Prototype

```
void IP_WEBS_OnConnectionLimit( const IP_WEBS_API * pIP_API,
                                      void        * CtrlSock );
```

Parameter

Parameter	Description
pIP_API	[IN] Pointer to a structure of type IP_FTPS_API.
CtrlSock	[IN] Pointer to the socket which is related to the command connection.

Table 15.6: IP_WEBS_OnConnectionLimit() parameter list

Additional information

This function is typically called by the application if the connection limit is reached. The structure type IP_WEBS_API contains mappings of the required socket functions to the actual IP stack. This is required because the socket functions are slightly different on different systems. Refer to *IP_WEBS_Process()* on page 287 and *IP_WEBS_ProcessLast()* on page 288 for further information.

Example

Pseudo code:

```
//
// Call IP_WEBS_Process() or IP_WEBS_ProcessLast() if multiple or just
// one more connection is available
//
do {
  if (NumAvailableConnections > 1) {
    IP_WEBS_Process();
    return;
  } else if (NumAvailableConnections == 1) {
    IP_WEBS_ProcessLast();
    return;
  }
  Delay();
} while (!Timeout)
//
// No connection available even after waiting => Output error message
//
IP_WEBS_OnConnectionLimit();
```


15.11.4 IP_WEBS_SendMem()

Description

Sends data to a connected target.

Prototype

```
int  IP_WEBS_SendMem   (      WEBS_OUTPUT * pOutput,
                        const char        * s,
                              unsigned      NumBytes);
```

Parameter

Parameter	Description
pOutput	[IN] Pointer to the WEBS_OUTPUT structure.
s	[IN] Pointer to a memory location that should be transmitted.
NumBytes	[IN] Number of bytes that should be sent.

Table 15.7: IP_WEBS_SendMem() parameter list

Return value

0 OK.

15.11.5 IP_WEBS_SendString()

Description

Sends a zero-terminated string to a connected target.

Prototype

```
int  IP_WEBS_SendString(       WEBS_OUTPUT * pOutput,
                         const char        * s);
```

Parameter

Parameter	Description
pOutput	[IN] Pointer to the WEBS_OUTPUT structure.
s	[IN] Pointer to a string that should be transmitted.

Table 15.8: IP_WEBS_SendString() parameter list

Return value

0 OK.

15.11.6 IP_WEBS_SendStringEnc()

Description

Encodes and sends a zero-terminated string to a connected target.

Prototype

```
int  IP_WEBS_SendString(       WEBS_OUTPUT * pOutput,
                         const char        * s);
```

Parameter

Parameter	Description
pOutput	[IN] Pointer to the WEBS_OUTPUT structure.
s	[IN] Pointer to a string that should be transmitted.

Table 15.9: IP_WEBS_SendStringEnc() parameter list

Return value

0 OK.

Additional information

This function encodes the string s with URL encoding, which means that spaces are changed into "+" and special characters are encoded to hexadecimal values. Refer to *[RFC 1738]* for detailed information about URL encoding.

15.11.7 IP_WEBS_SendUnsigned()

Description

Sends an unsigned value to the client.

Prototype

```
int  IP_WEBS_SendUnsigned ( WEBS_OUTPUT * pOutput,
                            unsigned      v,
                            unsigned      Base,
                            int           NumDigits );
```

Parameter

Parameter	Description
pOutput	[IN] Pointer to the WEBS_OUTPUT structure.
s	[IN] Value that should be sent.
Base	[IN] Numerical base.
NumDigits	[IN] Number of digits that should be sent. 0 can be used as a wild-card.

Table 15.10: IP_WEBS_SendUnsigned() parameter list

Return value

0 OK.

15.11.8 IP_WEBS_SetFileInfoCallback()

Description

Sets a callback function to receive the file information which are used by the stack.

Prototype

```
void IP_WEBS_SetFileInfoCallback ( IP_WEBS_pfGetFileInfo pf );
```

Parameter

Parameter	Description
pf	[IN] Pointer to a callback function.

Table 15.11: IP_WEBS_SetFileInfoCallback() parameter list

Additional information

The function can be used to change the default behavior of the web server. If the file info callback function is set, the web server calls it to retrieve the file information. The file information are used to decide how to handle the file and to build the HTML header. By default (no file info callback function is set), the web server parses every file with the extension `.htm` to check if dynamic content is included; all requested files with the extension `.cgi` are recognized as virtual files. Beside of that, the web server sends by default the expiration date of a web site in the HTML header. The default expiration date (THU, 01 JAN 1995 00:00:00 GMT) is in the past, so that the requested website will never be cached. This is a reasonable default for web pages with dynamic content. If the callback function returns 0 for DateExp, the expiration date will not be included in the header. For static websites, it is possible to add the optional "Last-Modified" header field. The "Last-Modified" header field is not part of the header by default. Refer to *Structure IP_WEBS_FILE_INFO* on page 314 for detailed information about the structure `IP_WEBS_FILE_INFO`.

Example

```
static void _GetFileInfo(const char * sFilename, IP_WEBS_FILE_INFO * pFileInfo){
  int v;

  //
  // .cgi files are virtual, everything else is not
  //
  v = IP_WEBS_CompareFilenameExt(sFilename, ".cgi");
  pFileInfo->IsVirtual = v ? 0 : 1;
  //
  // .htm files contain dynamic content, everything else is not
  //
  v = IP_WEBS_CompareFilenameExt(sFilename, ".htm");
  pFileInfo->AllowDynContent = v ? 0 : 1;
  //
  // If file is a virtual file or includes dynamic content,
  // get current time and date stamp as file time
  //
  pFileInfo->DateLastMod = _GetTimeDate();
  if (pFileInfo->IsVirtual || pFileInfo->AllowDynContent) {
    //
    // Set last-modified and expiration time and date
    //
    pFileInfo->DateExp     = _GetTimeDate(); // If "Expires" HTTP header field should
                                             // be transmitted, set expiration date.
  } else {
    pFileInfo->DateExp     = 0xEE210000; // Expiration far away (01 Jan 2099)
                                         // if content is static
  }
}
```


15.11.9 IP_WEBS_AddFileTypeHook()

Description

Registers an element of type WEBS_FILE_TYPE_HOOK to extend or override the list of file extension to MIME type correlation.

Prototype

```
void IP_WEBS_AddFileTypeHook ( WEBS_FILE_TYPE_HOOK * pHook,
                               const char          * sExt,
                               const char          * sContent );
```

Parameter

Parameter	Description
pHook	[IN] Pointer to an element of type WEBS_FILE_TYPE_HOOK.
sExt	[IN] String containing the extension without leading dot.
sContent	[IN] String containing the MIME type associated to the extension.

Table 15.12: IP_WEBS_AddFileTypeHook() parameter list

Additional information

The function can be used to extend or override the basic list of file extension to MIME type correlations included in the Web server. It might be necessary to extend the this list in case you want to serve a yet unkown file format. The header sent for this file in case a client reuqests it will be generated based on this information. Refer to *Structure WEBS_FILE_TYPE_HOOK* on page 318 for detailed information about the structure WEBS_FILE_TYPE_HOOK.

Example

```
static WEBS_FILE_TYPE_HOOK _FileTypeHook;

int main(void){
  //
  // Register *.new files to be treated as binary that will
  // be offered to be downloaded by the browser.
  //
  IP_WEBS_AddFileTypeHook(&_FileTypeHook, "new", "application/octet-stream");
}
```


15.11.10IP_WEBS_ConfigSendVFileHeader()

Description

Configures behavior of automatically sending a header containing a MIME type associated to the requested files extension based on an internal list for a requested virtual file.

Prototype

```
void IP_WEBS_ConfigSendVFileHeader ( U8 OnOff );
```

Parameter

Parameter	Description
OnOff	[IN] 0: Off, header will not be automatically generated and sent. 1: On, header will be automatically generated. Default: On.

Table 15.13: IP_WEBS_ConfigSendVFileHeader() parameter list

Additional information

In case you decide not to let the Web server generate a header with the best content believed to be known you will either have to completely send a header on your own or sending a header using the function *IP_WEBS_SendHeader()* on page 298. Sending a header has to be done before sending any other content.

15.11.11IP_WEBS_ConfigSendVFileHookHeader()

Description

Configures behavior of automatically sending a header containing a MIME type associated to the requested files extension based on an internal list for a requested file being served by a registered VFile hook.

Prototype

```
void IP_WEBS_ConfigSendVFileHookHeader ( U8 OnOff );
```

Parameter

Parameter	Description
OnOff	[IN] 0: Off, header will not be automatically generated and sent. 1: On, header will be automatically generated. Default: On.

Table 15.14: IP_WEBS_ConfigSendVFileHookHeader() parameter list

Additional information

In case you decide not to let the Web server generate a header with the best content believed to be known you will either have to completely send a header on your own or sending a header using the function *IP_WEBS_SendHeader()* on page 298. Sending a header has to be done before sending any other content.

15.11.12IP_WEBS_SendHeader()

Description

Generates and sends a header based on the information passed to this function.

Prototype

```
void IP_WEBS_SendHeader ( WEBS_OUTPUT * pContext,
                          const char  * sFileName,
                          const char  * sMimeType );
```

Parameter

Parameter	Description
pContext	[IN] Pointer to the context used for sending data from your callback to the client.
sFileName	[IN] String containing the file name including extension to be written to the header.
sMimeType	[IN] String containing the MIME type that is sent back in the header.

Table 15.15: IP_WEBS_SendHeader() parameter list

Additional information

This function can be used in case automatically generating and sending a header has been switched off using *IP_WEBS_ConfigSendVFileHeader()* on page 296 or *IP_WEBS_ConfigSendVFileHookHeader()* on page 297. Typically this is the first function you call from your callback generating content for a virtual file or a VFile hook registered callback providing content before you send any other data.

15.11.13IP_WEBS_CompareFileNameExt()

Description

Checks if the given filename has the given extension.

Prototype

```
char IP_WEBS_CompareFilenameExt( const char * sFilename,
                                 const char * sExt );
```

Parameter

Parameter	Description
sFilename	[IN] Name of the file.
sExt	[IN] Extension which should be checked.

Table 15.16: IP_WEBS_CompareFilenameExt() parameter list

Return value

0 Match
!= 0 Mismatch

Additional information

The test is case-sensitive, meaning:

```
IP_WEBS_CompareFilenameExt("Index.html", ".html")     ---> Match
IP_WEBS_CompareFilenameExt("Index.htm",  ".html")     ---> Mismatch
IP_WEBS_CompareFilenameExt("Index.HTML", ".html")     ---> Mismatch
IP_WEBS_CompareFilenameExt("Index.html", ".HTML")     ---> Mismatch
```


15.11.14IP_WEBS_GetNumParas()

Description

Returns the number of parameter/value pairs.

Prototype

```
int  IP_WEBS_GetNumParas ( const char * sParameters );
```

Parameter

Parameter	Description
sParameters	[IN] Zero-terminated string with parameter/value pairs.

Table 15.17: IP_WEBS_GetNumParas() parameter list

Return value

Number of parameters/value pairs.
-1 if the string does not include parameter value pairs.

Additional information

Parameters are separated from values by a '='. If a string includes more as one parameter/value pair, the parameter/value pairs are separated by a '&'. For example, if the virtual file Send.cgi gets two parameters, the string should be similar to the following: Send.cgi?FirstName=Foo&LastName=Bar

`sParameter` is in this case `FirstName=Foo&LastName=Bar`. If you call `IP_WEBS_GetNumParas()` with this string, the return value will be 2.

15.11.15IP_WEBS_GetParaValue()

Description

Parses a string for valid parameter/value pairs and writes the results in the respective buffers.

Prototype

```
int IP_WEBS_GetParaValue( const char * sBuffer,
                                  int    ParaNum,
                                  char * sPara,
                                  int    ParaLen,
                                  char * sValue,
                                  int    ValueLen );
```

Parameter

Parameter	Description
sBuffer	[IN] Zero-terminated parameter/value string that should be parsed.
ParaNum	[IN] Zero-based index of the parameter/value pairs.
sPara	[Out] Buffer to store the the parameter name. (Optional, can be NULL.)
ParaLen	[IN] Size of the buffer to store the parameter. (0 if sPara is NULL.)
sValue	[Out] Buffer to store the the value. (Optional, can be NULL.)
ValueLen	[IN] Size of the buffer to store the value. (0 if sValue is NULL.)

Table 15.18: IP_WEBS_GetParaValue() parameter list

Return value

0: O.K.
>0: Error

Additional information

A valid string is in the following format:
<Param0>=<Value0>&<Param1>=<Value1>& ... <Paramn>=<Valuen>

If the parameter value string is FirstName=John&LastName=Doo and parameter 0 should be copied, sPara will be FirstName and sValue John. If parameter 1 should be copied, sPara will be LastName and sValue Doo.

15.11.16IP_WEBS_GetParaValuePtr()

Description

Parses a string for valid parameter/value pairs and returns a pointer to the requested paramater and the length of the paramater string without termination.

Prototype

```
int IP_WEBS_GetParaValuePtr( const char *  sBuffer,
                                   int     ParaNum,
                             const char ** ppPara,
                                   int   *  pParaLen,
                             const char ** ppValue,
                                   int   *  pValueLen );
```

Parameter

Parameter	Description
sBuffer	[IN] Zero-terminated parameter/value string that should be parsed.
ParaNum	[IN] Zero-based index of the parameter/value pairs.
ppPara	[Out] Pointer to the pointer locating the start of the requested paramater name. (Optional, can be NULL.)
pParaLen	[OUT] Pointer to a buffer to store the length of the parameter name without termination. (Optional, can be NULL.)
ppValue	[Out] Pointer to the pointer locating the start of the requested parameter value. (Optional, can be NULL.)
pValueLen	[OUT] Pointer to a buffer to store the length of the parameter value without termination. (Optional, can be NULL.)

Table 15.19: IP_WEBS_GetParaValuePtr() parameter list

Return value

0: O.K.
>0: Error

Additional information

A valid string is in the following format:
<Param0>=<Value0>&<Param1>=<Value1>& ... <Paramn>=<Valuen>

This function can be used in case you simply want to check or use the paramaters passed by the client without modyfing them. Depending on your application this might save you a lot of stack that otherwise would have to be wasted for copying the same data that is already perfectly present to another location. This saves execution time as of course the data will not have to be copied.

Example

```
/* Excerpt from OS_IP_Webserver.c */
/*********************************************************************
*
*       _callback_CGI_Send
*/
static void _callback_CGI_Send(WEBS_OUTPUT * pOutput, const char * sParameters) {
        int    r;
  const char * pFirstName;
        int    FirstNameLen;
  const char * pLastName;
        int    LastNameLen;

    IP_WEBS_SendString(pOutput,  "<HTML><HEAD><TITLE>Virtual  file  example</TITLE></
HEAD>");
  IP_WEBS_SendString(pOutput, "<style type=\"text/css\"> \
    H1, H2, H3, H4 { color: white; font-family: Helvetica; } \
```



```
    PRE {  color: white;  margin-left: 2%; ; font-size=150%} \
    BODY{padding:0px; margin:0px; text-align:center; font-family:Verdana, Helvetica,
sans-serif; background:#6699CC url(bg.png) repeat-x; font-size:11px; color:white } \
    A:link { font-weight:bold; color:white; text-decoration:none; } \
    A:visited { font-weight:bold; color:silver; text-decoration:none; } \
    A:focus { font-weight:bold; color:white; text-decoration:underline; } \
    A:hover { font-weight:bold; color:silver; text-decoration:none; } \
    A:active { font-weight:bold; color:white; text-decoration:underline; }\
    </style>");
  IP_WEBS_SendString(pOutput, "<BODY><CENTER><HR><H2>Virtual file example</H2><HR></
CENTER><BR><BR><BR>");
  r  = IP_WEBS_GetParaValuePtr(sParameters, 0, NULL, 0, &pFirstName, &FirstNameLen);
  r |= IP_WEBS_GetParaValuePtr(sParameters, 1, NULL, 0, &pLastName, &LastNameLen);
  if (r == 0) {
    IP_WEBS_SendString(pOutput, "First name: ");
    IP_WEBS_SendMem(pOutput, pFirstName, FirstNameLen);
    IP_WEBS_SendString(pOutput, "<BR>Last name: ");
    IP_WEBS_SendMem(pOutput, pLastName, LastNameLen);
  } else {
    IP_WEBS_SendString(pOutput, "<BR>Error!");
  }
  IP_WEBS_SendString(pOutput, "<BR><BR><BR>");
   IP_WEBS_SendString(pOutput, "<HR><CENTER><A HREF=\"index.htm\">Back to main</A></
CENTER><IMG SRC=\"logo.gif\" ALT=\"Segger logo\">   SEGGER Microcontroller
GmbH &amp; Co. KG   <A HREF=\"http://www.segger.com\">www.segger.com</A></
BODY></HTML>");
}
```


15.11.17IP_WEBS_DecodeAndCopyStr()

Description

Checks if a string includes url encoded characters, decodes the characters and copies them into destination buffer.

Prototype

```
void IP_WEBS_DecodeAndCopyStr (       char * pDest,
                                      int    DestLen,
                                const char * pSrc,
                                      int    SrcLen );
```

Parameter

Parameter	Description
pDest	[OUT] Buffer to store the decoded string.
DestLen	[IN] Size of the destination buffer.
pSrc	[IN] Source string that should be decoded.
SrcLen	[IN] Size of the source string.

Table 15.20: IP_WEBS_DecodeAndCopyStr() parameter list

Additional information

Destination string is 0-terminated. Source and destination buffer can be identical.

pSrc	SrcLen	pDest	DestLen
"FirstName=J%F6rg"	16	"FirstName=Jörg\0"	15
"FirstName=John"	14	"FirstName=John\0"	15

Table 15.21: Example

15.11.18IP_WEBS_DecodeString()

Description

Checks if a string includes url encoded characters, decodes the characters.

Prototype

```
int IP_WEBS_DecodeString( const char * s );
```

Parameter

Parameter	Description
s	[IN/OUT] Zero-terminated string that should be decoded.

Table 15.22: IP_WEBS_DecodeString() parameter list

Return value

0 String does not include url encoded characters. No change.
>0 Length of the decoded string, including the terminating null character.

15.11.19IP_WEBS_AddVFileHook()

Description

Registers a function table containing callbacks to check and serve simple virtual file content that is not further processed by the Web server.

Prototype

```
void IP_WEBS_AddVFileHook ( WEBS_VFILE_HOOK        * pHook,
                            WEBS_VFILE_APPLICATION * pVFileApp );
```

Parameter

Parameter	Description
pHook	[IN] Pointer to an element of type WEBS_VFILE_HOOK.
pVFileApp	[IN] Pointer to an element of type WEBS_VFILE_APPLICATION.

Table 15.23: IP_WEBS_AddVFileHook() parameter list

Additional information

The function can be used to serve simple dynamically generated content for a requested file name that is simply sent back as generated by the application and is not further processed by the Web server. Refer to *Structure WEBS_VFILE_HOOK* on page 316 for detailed information about the structure WEBS_VFILE_HOOK. Refer to *Structure WEBS_VFILE_APPLICATION* on page 315 for detailed information about the structure WEBS_VFILE_APPLICATION.

Example

```
/* Excerpt from OS_IP_WebserverUPnP.c */
/*********************************************************************
*
*       _UPnP_GenerateSend_upnp_xml
*
* Function description
*   Send the content for the requested file using the callback provided.
*
* Parameters
*   pContextIn     - Send context of the connection processed for
*                    forwarding it to the callback used for output.
*   pf             - Function pointer to the callback that has to be
*                    for sending the content of the VFile.
*     pContextOut    - Out context of the connection processed.
*     pData          - Pointer to the data that will be sent
*     NumBytes       - Number of bytes to send from pData. If NumBytes
*                      is passed as 0 the send function will run a strlen()
*                      on pData expecting a string.
*
* Notes
*   (1) The data does not need to be sent in one call of the callback
*       routine. The data can be sent in blocks of data and will be
*       flushed out automatically at least once returning from this
*       routine.
*/
static void _UPnP_GenerateSend_upnp_xml(void * pContextIn, void (*pf) (void * pCon-
textOut, const char * pData, unsigned NumBytes)) {
  char ac[128];

  pf(pContextIn, "<?xml version=\"1.0\"?>\r\n"
                 "<root xmlns=\"urn:schemas-upnp-org:device-1-0\">\r\n"
                   "<specVersion>\r\n"
                     "<major>1</major>\r\n"
                     "<minor>0</minor>\r\n"
                     "</specVersion>\r\n", 0);
}
```



```
/* Excerpt from OS_IP_WebserverUPnP.c */
//
// UPnP webserver VFile hook
//
static WEBS_VFILE_HOOK _UPnP_VFileHook;

/* Excerpt from OS_IP_WebserverUPnP.c */
/*********************************************************************
*
*       _UPnP_CheckVFile
*
* Function description
*   Check if we have content that we can deliver for the requested
*   file using the VFile hook system.
*
* Parameters
*   sFileName      - Name of the file that is requested
*   pIndex         - Pointer to a variable that has to be filled with
*                    the index of the entry found in case of using a
*                    filename<=>content list.
*                    Alternative all comparisons can be done using the
*                    filename. In this case the index is meaningless
*                    and does not need to be returned by this routine.
*
* Return value
*   0              - We do not have content to send for this filename,
*                    fall back to the typical methods for retrieving
*                    a file from the web server.
*   1              - We have content that can be sent using the VFile
*                    hook system.
*/
static int _UPnP_CheckVFile(const char * sFileName, unsigned * pIndex) {
  unsigned i;

  //
  // Generated VFiles
  //
  if (strcmp(sFileName, "/upnp.xml") == 0) {
    return 1;
  }
  //
  // Static VFiles
  //
  for (i = 0; i < SEGGER_COUNTOF(_VFileList); i++) {
    if (strcmp(sFileName, _VFileList[i].sFileName) == 0) {
      *pIndex = i;
      return 1;
    }
  }
  return 0;
}

/*********************************************************************
*
*       _UPnP_SendVFile
*
* Function description
*   Send the content for the requested file using the callback provided.
*
* Parameters
*   pContextIn     - Send context of the connection processed for
*                    forwarding it to the callback used for output.
*   Index          - Index of the entry of a filename<=>content list
*                    if used. Alternative all comparisons can be done
*                    using the filename. In this case the index is
*                    meaningless. If using a filename<=>content list
*                    this is faster than searching again.
```



```
*   sFileName      - Name of the file that is requested. In case of
*                    working with the Index this is meaningless.
*   pf             - Function pointer to the callback that has to be
*                    for sending the content of the VFile.
*     pContextOut    - Out context of the connection processed.
*     pData          - Pointer to the data that will be sent
*     NumBytes       - Number of bytes to send from pData. If NumBytes
*                      is passed as 0 the send function will run a strlen()
*                      on pData expecting a string.
*/
static void _UPnP_SendVFile(void * pContextIn, unsigned Index, const char * sFile-
Name, void (*pf) (void * pContextOut, const char * pData, unsigned NumBytes)) {
  (void)sFileName;

  //
  // Generated VFiles
  //
  if (strcmp(sFileName, "/upnp.xml") == 0) {
    _UPnP_GenerateSend_upnp_xml(pContextIn, pf);
    return;
  }
  //
  // Static VFiles
  //
  pf(pContextIn, _VFileList[Index].pData, _VFileList[Index].NumBytes);
}

static WEBS_VFILE_APPLICATION _UPnP_VFileAPI = {
  _UPnP_CheckVFile,
  _UPnP_SendVFile
};

/* Excerpt from OS_IP_WebserverUPnP.c */

/*********************************************************************
*
*       MainTask
*/
void MainTask(void);
void MainTask(void) {
  //
  // Activate UPnP with VFile hook for needed XML files
  //
  IP_WEBS_AddVFileHook(&_UPnP_VFileHook, &_UPnP_VFileAPI);
}
```


15.11.20IP_UTIL_BASE64_Decode()

Description

Performs BASE-64 decoding according to RFC3548.

Prototype

```
int IP_UTIL_BASE64_Decode( const U8  * pSrc,
                                 int   SrcLen,
                                 U8  * pDest,
                                 int * pDestLen );
```

Parameter

Parameter	Description
pSrc	[IN] Pointer to data to encode.
SrcLen	Number of bytes to encode.
pDest	[IN] Pointer to the destination buffer.
pDestLen	[IN] Pointer to the destination buffer size. [OUT] Pointer to the number of bytes used in the destination buffer.

Table 15.24: IP_UTIL_BASE64_Decode() parameter list

Return value

< 0	Error
> 0	Number of source bytes encoded, further call required
0	All bytes encoded

Additional information

For more information, refer to *http://tools.ietf.org/html/rfc3548.*

15.11.21IP_UTIL_BASE64_Encode()

Description

Performs BASE-64 encoding according to RFC3548.

Prototype

```
int IP_UTIL_BASE64_Encode( const U8  * pSrc,
                                 int   SrcLen,
                                 U8  * pDest,
                                 int * pDestLen );
```

Parameter

Parameter	Description
pSrc	[IN] Pointer to data to encode.
SrcLen	Number of bytes to encode.
pDest	[IN] Pointer to the destination buffer.
pDestLen	[IN] Pointer to the destination buffer size. [OUT] Pointer to the number of bytes used in the destination buffer.

Table 15.25: IP_UTIL_BASE64_Encode() parameter list

Return value

< 0	Error
> 0	Number of source bytes encoded, further call required
0	All bytes encoded

Additional information

For more information, refer to *http://tools.ietf.org/html/rfc3548.*

15.12 Web server data structures

15.12.1 Structure WEBS_CGI

Description

Used to store the CGI command names and the pointer to the proper callback functions.

Prototype

```
typedef struct {
  const char * sName;
  void  (*pf)(WEBS_OUTPUT * pOutput, const char * sParameters);
} WEBS_CGI;
```

Member	Description
sName	Name of the CGI command.
pf	Pointer to a callback function.

Table 15.26: Structure WEBS_CGI member list

Additional information

Refer to *Common Gateway Interface (CGI)* on page 270 for detailed information about the use of this structure.

15.12.2 Structure WEBS_ACCESS_CONTROL

Description

Used to store information for the HTTP Basic Authentication scheme.

Prototype

```
typedef struct {
  const char * sPath;
  const char * sRealm;
  const char * sUserPass;
} WEBS_ACCESS_CONTROL;
```

Member	Description
sPath	A string which defines the path of the resources.
sRealm	A string which defines the realm which requires authentication. Optional, can be NULL.
sUserPass	A string containing the user name/password combination. Optional, can be NULL.

Table 15.27: Structure WEBS_ACCESS_CONTROL member list

Additional information

If sRealm is initialized with NULL, sUserPass is not interpreted by the web server. Refer to *Authentication* on page 274 for detailed information about the HTTP Basic Authentication scheme.

15.12.3 Structure WEBS_APPLICATION

Description

Used to store application-specific parameters.

Prototype

```
typedef struct {
  const WEBS_CGI * paCGI;
  WEBS_ACCESS_CONTROL * paAccess;
  void  (*pfHandleParameter)(       WEBS_OUTPUT * pOutput,
                             const char           sPara,
                             const char         * sValue );

} WEBS_APPLICATION;
```

Member	Description
paCGI	Pointer to an array of structures of type WEBS_CGI.
paAccess	Pointer to an array of structures of type WEBS_ACCESS_CONTROL.
pfHandleParameter	Pointer to an array of structures of type WEBS_CGI.

Table 15.28: Structure WEBS_APPLICATION member list

15.12.4 Structure IP_WEBS_FILE_INFO

Description

Used to store application-specific parameters.

Prototype

```
typedef struct {
  U32 DateLastMod;         // Used for "Last modified" header field
  U32 DateExp;             // Used for "Expires" header field
  U8  IsVirtual;
  U8  AllowDynContent;
} IP_WEBS_FILE_INFO;
```

Member	Description
DateLastModified	The date when the file has been last modified.
DateExp	The date of the expiration of the valiity.
IsVirtual	Flag to indicate if a file is virtual or not. Valid values are 0 for non-virtual, 1 for virtual files.
AllowDynContent	Flag to indicate if a file should be parsed for dynamic content or not. 0 means that the file should not be parsed for dynamic content, 1 means that the file should be parsed for dynamic content.

Table 15.29: Structure IP_WEBS_FILE_INFO member list

15.12.5 Structure WEBS_VFILE_APPLICATION

Description

Used to check if the application can provide content for a simple VFile.

Prototype

```
typedef struct WEBS_VFILE_APPLICATION {
  int  (*pfCheckVFile)(const char * sFileName, unsigned * pIndex);
  void (*pfSendVFile) (void * pContextIn,
                       unsigned Index,
                       const char * sFileName,
                       void (*pf) (void * pContextOut,
                                   const char * pData,
                                   unsigned NumBytes));
} WEBS_VFILE_APPLICATION;
```

Member	Description
pfCheckVFile	Pointer to a callback for checking if content for a requested file name can be served.
pfSendVFile	Pointer to a callback for actually sending the content for the requested file name using the provided callback pf. In case NumBytes is passed with '0' the callback expects to find a string and will automatically run strlen() to find out the length of the string internally. In case NumBytes is not passed '>0' only NumBytes from the start of pData will be sent.

Table 15.30: Structure WEBS_VFILE_APPLICATION member list

15.12.6 Structure WEBS_VFILE_HOOK

Description

Used to send application generated content from the application upon request of a specific file name.

Prototype

```
typedef struct WEBS_VFILE_HOOK {
  struct WEBS_VFILE_HOOK        * pNext;
         WEBS_VFILE_APPLICATION * pVFileApp;
} WEBS_VFILE_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_VFILE_HOOK.
pVFileApp	Pointer to an element of type WEBS_VFILE_APPLICATION.

Table 15.31: Structure WEBS_VFILE_HOOK member list

Additional information

Refer to *Structure WEBS_VFILE_HOOK* on page 316 for detailed information about the structure WEBS_VFILE_HOOK. Refer to *Structure WEBS_VFILE_APPLICATION* on page 315 for detailed information about the structure WEBS_VFILE_APPLICATION.

15.12.7 Structure WEBS_FILE_TYPE

Description

Used to extend or overwrite the file extension to MIME type correlation.

Prototype

```
typedef struct WEBS_FILE_TYPE {
  const char *sExt;
  const char *sContent;
} WEBS_FILE_TYPE;
```

Member	Description
sExt	String containing the extension without leading dot.
sContent	String containing the MIME type associated to the extension.

Table 15.32: Structure WEBS_FILE_TYPE member list

15.12.8 Structure WEBS_FILE_TYPE_HOOK

Description

Used to extend or overwrite the file extension to MIME type correlation.

Prototype

```
typedef struct WEBS_FILE_TYPE_HOOK {
  struct WEBS_FILE_TYPE_HOOK * pNext;
         WEBS_FILE_TYPE        FileType;
} WEBS_FILE_TYPE_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_FILE_TYPE_HOOK.
FileType	Element of Structure WEBS_FILE_TYPE.

Table 15.33: Structure WEBS_VFILE_HOOK member list

Additional information

Refer to *Structure WEBS_FILE_TYPE_HOOK* on page 318 for detailed information about the structure WEBS_FILE_TYPE_HOOK. Refer to *Structure WEBS_FILE_TYPE* on page 317 for detailed information about the structure WEBS_FILE_TYPE.

15.13 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the web server presented in the tables below have been measured on an ARM7 and a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

Configuration used

```
#define WEBS_IN_BUFFER_SIZE        256
#define WEBS_OUT_BUFFER_SIZE       512
#define WEBS_TEMP_BUFFER_SIZE      256
#define WEBS_PARA_BUFFER_SIZE      256
#define WEBS_ERR_BUFFER_SIZE       128
#define WEBS_AUTH_BUFFER_SIZE      32
#define WEBS_FILENAME_BUFFER_SIZE  32
```

15.13.1 ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP Web server	approximately 5.9Kbyte

Table 15.34: Web server ROM usage ARM7

15.13.2 ROM usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

Addon	ROM
embOS/IP Web server	approximately 5.4Kbyte

Table 15.35: Web server ROM usage Cortex-M3

15.13.3 RAM usage:

Almost all of the RAM used by the web server is taken from task stacks. The amount of RAM required for every child task depends on the configuration of your server. The table below shows typical RAM requirements for your task stacks.

Task	Description	RAM
ParentTask	Listens for incoming connections.	approximately 500 bytes
ChildTask	Handles a request.	approximately 1800 bytes

Table 15.36: Web server RAM usage

Note: The web server requires at least 1 child task.

The approximately RAM usage for the web server can be calculated as follows:

RAM usage = 0.1 Kbytes + ParentTask + (NumberOfChildTasks * 1.8 Kbytes)

Example: Web server accepting only 1 connection

RAM usage = 0.1 Kbytes + 500 +(1 * 1.8 Kbytes)
RAM usage = 2.4 Kbytes

Example: Web server accepting up to 3 connections in parallel

RAM usage = 0.1 Kbytes + 500 + (3 * 1.8 Kbytes)
RAM usage = 6.0 Kbytes

Chapter 16

SMTP client (Add-on)

The embOS/IP SMTP client is an optional extension to embOS/IP. The SMTP client can be used with embOS/IP or with a different TCP/IP stack. All functions that are required to add the SMTP client task to your application are described in this chapter.

16.1 embOS/IP SMTP client

The embOS/IP SMTP client is an optional extension which can be seamlessly integrated into your TCP/IP application. It combines a maximum of performance with a small memory footprint. The SMTP client allows an embedded system to send emails with dynamically generated content. The RAM usage of the SMTP client module has been kept to a minimum by smart buffer handling.

The SMTP client implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 821]	Simple Mail Transfer Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc821.txt*
[RFC 974]	Mail routing and the domain system Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc974.txt*
[RFC 2554]	*SMTP Service Extension for Authentication* *Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc2554.txt*
[RFC 5321]	Simple Mail Transfer Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc5321.txt*

The following table shows the contents of the embOS/IP SMTP client root directory:

Directory	Content
`Application\`	Contains the example application to run the SMTP client with embOS/IP.
`Config`	Contains the SMTP client configuration file. Refer to *Configuration* on page 327 for detailed information.
`Inc`	Contains the required include files.
`IP`	Contains the SMTP client sources, `IP_SMTPC.c` and `IP_SMTPC.h`.
`Windows\SMTPC\`	Contains the source, the project files and an executable to run embOS/IP SMTP client on a Microsoft Windows host.

Supplied directory structure of embOS/IPSMTP client package

16.2 Feature list

- Low memory footprint.
- Independent of the TCP/IP stack: any stack with sockets can be used.
- Example applications included.
- Project for executable on PC for Microsoft Visual Studio included.

16.3 Requirements

TCP/IP stack

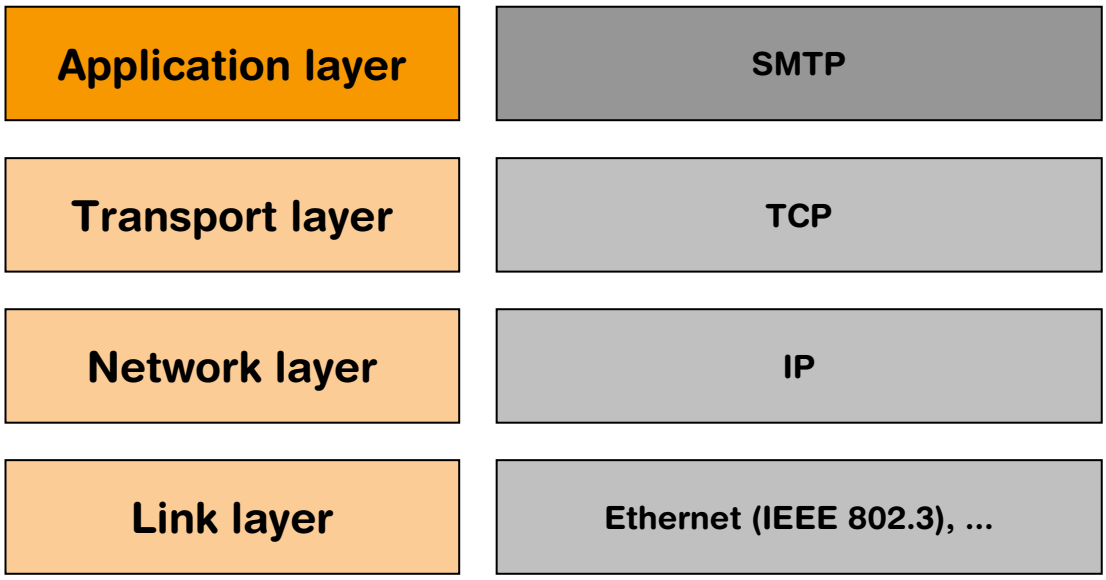
The embOS/IP SMTP client requires a TCP/IP stack. It is optimized for embOS/IP, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of embOS/IP.

Multi tasking

The SMTP client needs to run as a separate thread. Therefore, a multi tasking system is required to use the embOS/IP SMTP client.

16.4 SMTP backgrounds

The Simple Mail Transfer Protocol is a text based communication protocol for electronic mail transmission across IP networks.



Using SMTP, an embOS/IP application can transfer mail to an SMTP servers on the same network or to SMTP servers in other networks via a relay or gateway server accessible to both networks. When the embOS/IP SMTP client has a message to transmit, it establishes a TCP connection to an SMTP server and transmits after the handshaking the message content.

The handshaking mechanism includes normally an authentication process. The RFC's define the following four different authentication schemes:

- PLAIN
- LOGIN
- CRAM-MD5
- NTLM

In the current version, the embOS/IP SMTP client supports only PLAIN authentication. The following listing shows a typical SMTP session:

```
S:  220 srv.sample.com ESMTP
C:   HELO
S:  250 srv.sample.com
C:   AUTH LOGIN
S:  334 VXNlcm5hbWU6
C:   c3BzZXk29IulbkY29tZcZXIbtZ
S:  334 UGFzc3dvcmQ6
C:   UlblhFz7ZlblsZlZQ==
S:  235 go ahead
C:   Mail from:<user0@sample.com>
S:  250 ok
C:   Rcpt to:<user1@sample.com>
S:  250 ok
C:   Rcpt to:<user2@sample.com>
S:  250 ok
C:   Rcpt to:<user3@sample.com>
S:  250 ok
C:   DATA
S:  354 go ahead
C:   Message-ID: <1000.2234@sample.com>
C:   From: "User0" <User0@sample.com>
C:   TO: "User1" <User1@sample.com>
C:   CC: "User2" <User2@sample.com>, "User3" <User3@sample.com>
```



```
C:   Subject: Testmail
C:   Date: 1 Jan 2008 00:00 +0100
C:
C:   This is a test!
C:
C:   .
S:  250 ok 1231221612 qp 3364
C:   quit
S:  221 srv.sample.com
```


16.5 Configuration

The embOS/IP SMTP client can be used without changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which match the requirements of most applications.

The following types of configuration macros exist:

Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

Numerical values "N"

Numerical values are used somewhere in the source code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

Alias "A"

A macro which operates like a simple text substitute. An example would be the define `U8`, which the preprocessor would replace with `unsigned char`.

Function replacements "F"

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the source code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).

16.5.1 Compile time configuration switches

Type	Symbolic name	Default	Description
F	SMTPC_WARN	--	Defines a function to output warnings. In debug configurations (`DEBUG == 1`) `SMTPC_WARN` maps to `IP_Warnf_Application()`.
F	SMTPC_LOG	--	Defines a function to output logging messages. In debug configurations (`DEBUG == 1`) `SMTPC_LOG` maps to `IP_Logf_Application()`.
N	SMTPC_SERVER_PORT	25	Defines the port where the SMTP server is listening.
N	SMTPC_IN_BUFFER_SIZE	256	Defines the size of the input buffer. The input buffer is used to store the SMTP replies of the SMTP server.
N	SMTPC_AUTH_USER_BUFFER_SIZE	48	Defines the size of the buffer used for the Base-64 encoded user name.
N	SMTPC_AUTH_PASS_BUFFER_SIZE	48	Defines the size of the buffer used for the Base-64 encoded password.

16.6 API functions

Function	Description
SMTP client functions	
IP_SMTPC_Send()	Sends an email to a mail transfer agent.

Table 16.1: embOS/IP SMTP client interface function overview

16.6.1 IP_SMTPC_Send()

Description

Sends an email to one or multiple recipients.

Prototype

```
int IP_SMTPC_Send( const IP_SMTPC_API                * pIP_API,
                         IP_SMTPC_MAIL_ADDR          * paMailAddr,
                         int                           NumMailAddr,
                         IP_SMTPC_MESSAGE            * pMessage,
                         const IP_SMTPC_MTA          * pMTA,
                         const IP_SMTPC_APPLICATION * pApplication );
```

Parameter

Parameter	Description
pIP_API	[IN] Pointer to an IP_STMPC_API structure. Refer to *Structure IP_SMTPC_API* on page 331 for detailed information about the elements of the IP_SMTPC_API structure.
paMailAddr	[IN] Pointer to an array of IP_SMTPC_MAIL_ADDR structures. Refer to *Structure IP_SMTPC_MAIL_ADDR* on page 334 for detailed information about the elements of the IP_SMTPC_MAIL_ADDR structure. The first element of the array has to be filled with the data of the sender (FROM). The order of the following data sets for recipients (TO), carbon copies (CC) and blind carbon copies (BCC) is not relevant.
NumMailAddr	[IN] Number of email addresses.
pMessage	[IN] Pointer to an array of IP_SMTPC_MESSAGE structures. Refer to *Structure IP_SMTPC_MESSAGE* on page 336 for detailed information about the elements of the IP_SMTPC_MESSAGE structure.
pMTA	[IN] Pointer to an array of IP_SMTPC_MTA structures. Refer to *Structure IP_SMTPC_MTA* on page 337 for detailed information about the elements of the IP_SMTPC_MTA structure.
pApplication	[IN] Pointer to an array of IP_SMTPC_APPLICATION structures. Refer to *Structure IP_SMTPC_APPLICATION* on page 333 for detailed information about the elements of the IP_SMTPC_APPLICATION structure.

Table 16.2: IP_SMTPC_Send() parameter list

Return value

0 OK.
1 Error.

16.7 SMTP client data structures

Function	Description
IP_SMTPC_API	Structure with pointers to the required socket interface functions.
IP_SMTPC_APPLICATION	Structure with application related elements.
IP_SMTPC_MAIL_ADDR	Structure to store the mail addresses.
IP_SMTPC_MESSAGE	Structure defining the message format.
IP_SMTPC_MTA	Structure to store the login information for the mail transfer agent.

Table 16.3: embOS/IP SMTP client interface function overview

16.7.1 Structure IP_SMTPC_API

Description

Structure with pointers to the required socket interface functions.

Prototype

```
typedef struct {
  SMTPC_SOCKET (*pfConnect)    (char * SrvAddr);
  void         (*pfDisconnect) (SMTPC_SOCKET Socket);
  int          (*pfSend)       (const char *       pData,
                                int           Len,
                                SMTPC_SOCKET Socket);

  int          (*pfReceive)    (char *       pData,
                                int           Len,
                                SMTPC_SOCKET Socket);
} IP_SMTPC_API;
```

Member	Description
pfConnect	Pointer to the function (for example, `connect()`).
pfDisconnect	Pointer to the disconnect function (for example, `closesocket()`.
pfSend	Pointer to a callback function (for example, `send()`).
pfDisconnect	Pointer to a callback function (for example, `recv()`).

Table 16.4: Structure IP_SMTPC_API member list

Example

```
/*********************************************************************
*
*       _Connect
*
*  Function description
*    Creates a socket and opens a TCP connection to the mail host.
*/
static SMTPC_SOCKET _Connect(char * SrvAddr) {
  long IP;
  long Sock;
  struct hostent * pHostEntry;
  struct sockaddr_in sin;
  int r;
  //
  // Convert host into mail host
  //
  pHostEntry = gethostbyname(SrvAddr);
  if (pHostEntry == NULL) {
    SMTPC_LOG(("gethostbyname failed: %s\r\n", SrvAddr));
    return NULL;
  }
  IP = *(unsigned*)(*pHostEntry->h_addr_list);
  //
  // Create socket and connect to mail server
  //
  Sock = socket(AF_INET, SOCK_STREAM, 0);
  if(Sock  == -1) {
    SMTPC_LOG(("Could not create socket!"));
    return NULL;
  }
  IP_MEMSET(&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_port = htons(SERVER_PORT);
  sin.sin_addr.s_addr = IP;
  r = connect(Sock, (struct sockaddr*)&sin, sizeof(sin));
  if(r == SOCKET_ERROR) {
```



```
    SMTPC_LOG(("\nSocket error :"));
    return NULL;
  }
  SMTPC_LOG(("APP: Connected.\r\n"));
  return (SMTPC_SOCKET)Sock;
}

/*********************************************************************
*
*       _Disconnect
*
*  Function description
*    Closes a socket.
*/
static void _Disconnect(SMTPC_SOCKET Socket) {
  closesocket((long)Socket);
}

/*********************************************************************
*
*       _Send
*
*  Function description
*    Sends data via socket interface.
*/
static int _Send(const char * buf, int len, void * pConnectionInfo) {
  return send((long)pConnectionInfo, buf, len, 0);
}

/*********************************************************************
*
*       _Recv
*
*  Function description
*    Receives data via socket interface.
*/
static int _Recv(char * buf, int len, void * pConnectionInfo) {
  return recv((long)pConnectionInfo, buf, len, 0);
}

static const IP_SMTPC_API _IP_Api = {
  _Connect,
  _Disconnect,
  _Send,
  _Recv
};
```


16.7.2 Structure IP_SMTPC_APPLICATION

Description

Structure with pointers to application related functions.

Prototype

```
typedef struct {
  U32 (*pfGetTimeDate) (void);
  int (*pfCallback)(int Stat, void *p);
  const char * sDomain;   // email domain
} IP_SMTPC_APPLICATION;
```

Member	Description
pfGetTimeDate	Pointer to the function which returns the current system time. Used to set the correct date and time of the email.
pfCallback	Pointer to status callback function. Can be NULL.
sDomain	Domain name. For example, sample.com. According to RFC 821 the maximum total length of a domain name or number is 64 characters.

Table 16.5: Structure IP_SMTPC_APPLICATION member list

Example

```
*********************************************************************
*
*       _GetTimeDate
*/
static U32 _GetTimeDate(void) {
  U32 r;
  U16 Sec, Min, Hour;
  U16 Day, Month, Year;

  Sec   = 0;         // 0 based.  Valid range: 0..59
  Min   = 0;         // 0 based.  Valid range: 0..59
  Hour  = 0;         // 0 based.  Valid range: 0..23
  Day   = 1;         // 1 based.  Means that 1 is 1.
                     //           Valid range is 1..31 (depending on month)
  Month = 1;         // 1 based.  Means that January is 1. Valid range is 1..12.
  Year  = 28;        // 1980 based. Means that 2008 would be 28.
  r   = Sec / 2 + (Min << 5) + (Hour  << 11);
  r  |= (U32)(Day + (Month << 5) + (Year  << 9)) << 16;
  return r;
}

*********************************************************************
*
*       _Application
*/
static const SMTPC_APPLICATION _Application = {
  _GetTimeDate,
  NULL,
  "sample.com"    // Your domain.
};
```


16.7.3 Structure IP_SMTPC_MAIL_ADDR

Description

Structure to store an email address.

Prototype

```
typedef struct {
  const char * sName;
  const char * sAddr;
  int Type;
} IP_SMTPC_MAIL_ADDR;
```

Member	Description
sName	Name of the recipient (for example, "Foo Bar"). Can be NULL.
sAddr	email address of the recipient (for example, "foo@bar.com").
Type	Type of the email address.

Table 16.6: Structure IP_SMTPC_MAIL_ADDR member list

Valid values for parameter Type

Value	Description
SMTPC_REC_TYPE_FROM	email address of the sender (FROM).
SMTPC_REC_TYPE_TO	email address of the recipient (TO).
SMTPC_REC_TYPE_CC	email address of a recipient which should get a carbon copy (CC) of the email.
SMTPC_REC_TYPE_BC	email address of a recipient which should get a blind carbon copy (BCC) of the email.

Additional information

The structure is used to store the data sets of the sender and all recipients. IP_SMTPC_Send() gets a pointer to an array of IP_SMTPC_MAIL_ADDR structures as parameter. The first element of these array has to be filled with the data of the sender (FROM). The order of the following data sets for Recipients (TO), Carbon Copies (CC) and Blind Carbon Copies (BCC) is not relevant. For detailed information about IP_SMTPC_Send() refer to *IP_SMTPC_Send()* on page 329.

Example

```
/*********************************************************************
*
*       Mailer
*/
static void _Mailer(void) {
  SMTPC_MAIL_ADDR MailAddr[4];
  SMTPC_MTA Mta;
  SMTPC_MESSAGE Message;

  IP_MEMSET(&MailAddr, 0, sizeof(MailAddr));
  //
  // Sender
  //
  MailAddr[0].sName = 0; // for example, "Your name";
  MailAddr[0].sAddr = 0; // for example, "user@foobar.com";
  MailAddr[0].Type  = SMTPC_REC_TYPE_FROM;
  //
  // Recipient(s)
  //
  MailAddr[1].sName = 0; // "Recipient";
  MailAddr[1].sAddr = 0; // "recipient@foobar.com";
  MailAddr[1].Type  = SMTPC_REC_TYPE_TO;
```



```
  MailAddr[2].sName = 0; // "CC Recp 1";
  MailAddr[2].sAddr = 0; // "cc1@foobar.com";
  MailAddr[2].Type  = SMTPC_REC_TYPE_CC;
  MailAddr[3].sName = 0; // "BCC Recp 1"
  MailAddr[3].sAddr = 0; // "bcc1@foobar.com";;
  MailAddr[3].Type  = SMTPC_REC_TYPE_BCC;
  //
  // Message
  //
  Message.sSubject = "SMTP message sent via embOS/IP SMTP client";
  Message.sBody    = "embOS/IP SMTP client - www.segger.com";
  //
  // Fill mail transfer agent structure
  //
  Mta.sServer = 0; // for example, "mail.foobar.com";
  Mta.sUser   = 0; // for example, "user@foobar.com";
  Mta.sPass   = 0; // for example, "password";
  //
  // Check if sample is configured!
  //
  if(Mta.sServer == 0) {
        SMTPC_WARN(("You have to enter valid SMTP server, sender and recipi-
ent(s).\r\n"));
    while(1);
  }
  //
  // Wait until link is up. This can take 2-3 seconds if PHY has been reset.
  //
  while (IP_IFaceIsReady() == 0) {
    OS_Delay(100);
  }
  SMTPC_Send(&_IP_Api, &MailAddr[0], 4, &Message, &Mta, &_Application);
  while(1);
}
```


16.7.4 Structure IP_SMTPC_MESSAGE

Description

Structure to store the subject and the text of the email.

Prototype

```
typedef struct {
  const char      * sSubject;
  const char      * sBody;
  int               MessageSize;
} IP_SMTPC_MESSAGE;
```

Member	Description
sSubject	Pointer to the string used as subject of the email.
sBody	Pointer to the string used as message of the email.
MessageSize	Size of the message.

Table 16.7: Structure IP_SMTPC_MESSAGE member list

16.7.5 Structure IP_SMTPC_MTA

Description

Structure to store the server address and the login information.

Prototype

```
typedef struct {
  const char * sServer;
  const char * sUser;
  const char * sPass;
} IP_SMTPC_MTA;
```

Member	Description
sServer	Server address (for example, “mail.foobar.com”).
sUser	Account user name (for example, “foo@bar.com”). Can be NULL.
sPass	Account password (for example, “password”). Can be NULL.

Table 16.8: Structure IP_SMTPC_MTA member list

Additional information

The parameters sUser and sPass have to be NULL if no authentication is used by the server. If sUser is set in the application code, the client tries to use authentication. This means that the client sends the AUTH LOGIN command to the server. If the server does not support authentication, he will return an error code and the client closes the session.

16.8 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the SMTP client presented in the tables below have been measured on an ARM7 and a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

Configuration used

```
#define SMTPC_OUT_BUFFER_SIZE   256
```

16.8.1 Resource usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

16.8.1.1 ROM usage

Addon	ROM
embOS/IP SMTP client	approximately 7.1Kbyte

Table 16.9: SMTPC client ROM usage ARM7

16.8.1.2 RAM usage

Addon	RAM
embOS/IP SMTP client (w/o task stack)	approximately 4.7Kbyte

Table 16.10: SMTPC client RAM usage ARM7

16.8.2 Resource usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

16.8.2.1 ROM usage

Addon	ROM
embOS/IP SMTP client	approximately 6.5Kbyte

Table 16.11: SMTPC client ROM usage Cortex-M3

16.8.2.2 RAM usage

Addon	RAM
embOS/IP SMTP client w/o task stack	approximately 4.7Kbyte

Table 16.12: SMTPC client RAM usage Cortex-M3

Chapter 17

FTP server (Add-on)

The embOS/IP FTP server is an optional extension to the TCP/IP stack. The FTP server can be used with embOS/IP or with a different TCP/IP stack. All functions which are required to add a FTP server task to your application are described in this chapter.

17.1 embOS/IP FTP server

The embOS/IP FTP server is an optional extension which adds the FTP protocol to the stack. FTP stands for File Transfer Protocol. It is the basic mechanism for moving files between machines over TCP/IP based networks such as the Internet. FTP is a client/server protocol, meaning that one machine, the client, initiates a file transfer by contacting another machine, the server and making requests. The server must be operating before the client initiates his requests. Generally a client communicates with one server at a time, while most servers are designed to work with multiple simultaneous clients.

The FTP server implements the relevant parts of the following RFCs.

RFC#	Description
[RFC 959]	FTP - File Transfer Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc959.txt*

The following table shows the contents of the embOS/IP FTP server root directory:

Directory	Contents
`Application\`	Contains the example application to run the FTP server with embOS/IP.
`Config`	Contains the FTP server configuration file.
`Inc`	Contains the required include files.
`IP`	Contains the FTP server sources.
`IP\FS\`	Contains the sources for the file system abstraction layer and the read-only file system. Refer to *File system abstraction layer function table* on page 457 for detailed information.
`Windows\FTPserver\`	Contains the source, the project files and an executable to run embOS/IP FTP server on a Microsoft Windows host.

Supplied directory structure of embOS/IP FTP server package

17.2 Feature list

- Low memory footprint.
- Multiple connections supported.
- Independent of the file system: Any file system can be used.
- Independent of the TCP/IP stack: Any stack with sockets can be used.
- Demo application included.
- Project for executable on PC for Microsoft Visual Studio included.

17.3 Requirements

TCP/IP stack

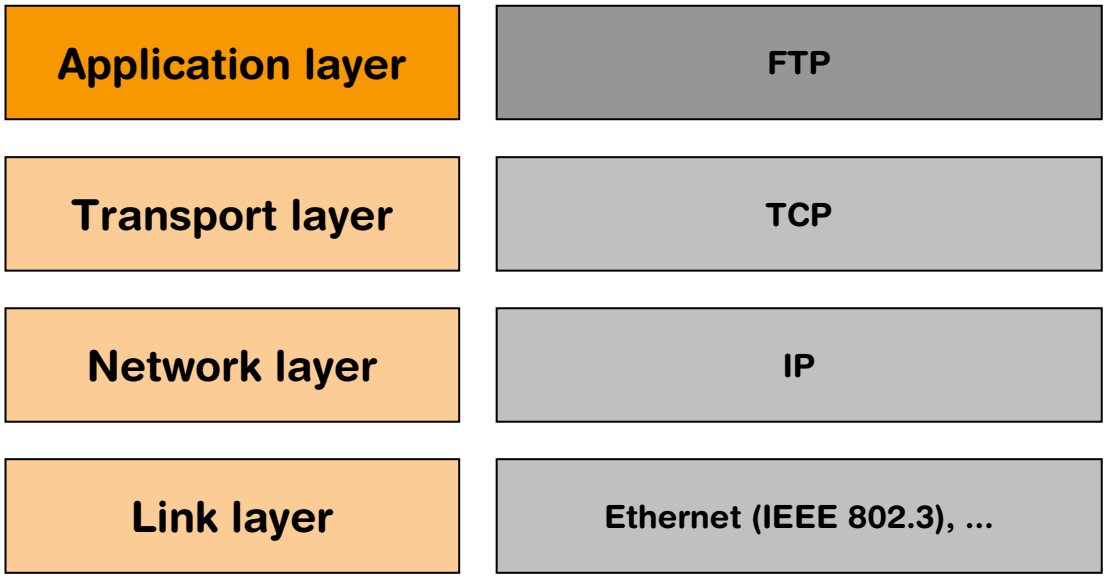
The embOS/IP FTP server requires a TCP/IP stack. It is optimized for embOS/IP, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of embOS/IP.

Multi tasking

The FTP server needs to run as a separate thread. Therefore, a multi tasking system is required to use the embOS/IP FTP server.

17.4 FTP basics

The File Transfer Protocol (FTP) is an application layer protocol. FTP is an unusual service in that it utilizes two ports, a 'Data' port and a 'CMD' (command) port. Traditionally these are port 21 for the command port and port 20 for the data port. FTP can be used in two modes, active and passive. Depending on the mode, the data port is not always on port 20.



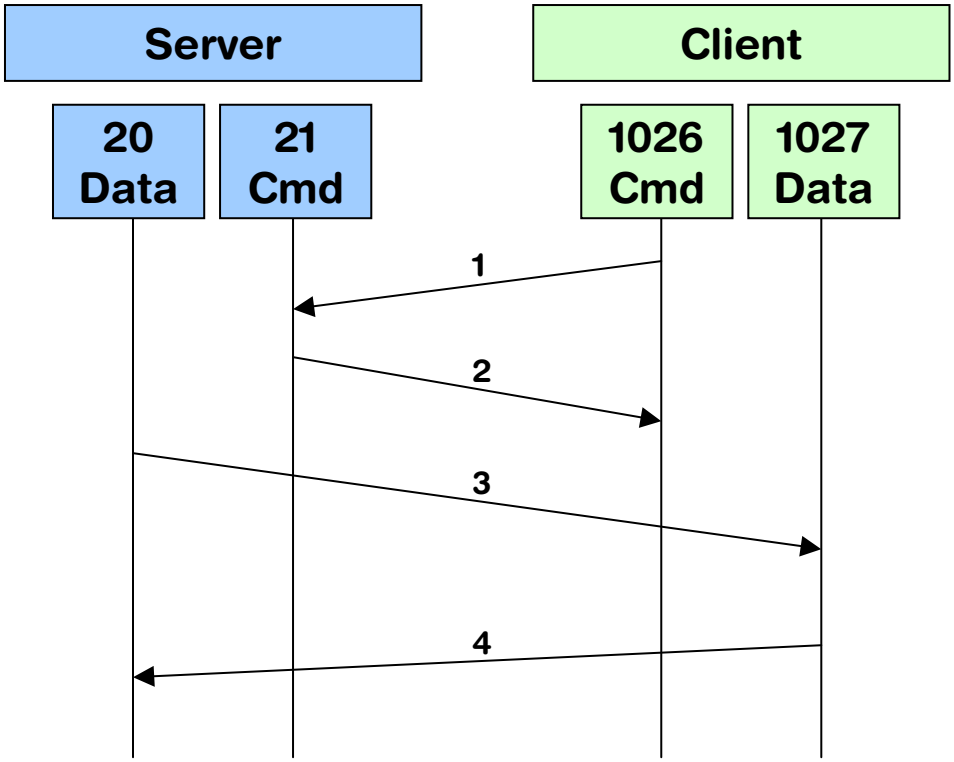
When an FTP client contacts a server, a TCP connection is established between the two machines. The server does a passive open (a socket is listen) when it begins operation; thereafter clients can connect with the server via active opens. This TCP connection persists for as long as the client maintains a session with the server, (usually determined by a human user) and is used to convey commands from the client to the server, and the server replies back to the client. This connection is referred to as the FTP command connection.

The FTP commands from the client to the server consist of short sets of ASCII characters, followed by optional command parameters. For example, the FTP command to display the current working directory is `PWD` (Print Working Directory). All commands are terminated by a carriage return-linefeed sequence (CRLF) (ASCII 10,13; or Ctrl-J, Ctrl-M). The servers replies consist of a 3 digit code (in ASCII) followed by some explanatory text. Generally codes in the 200s are success and 500s are failures. See the RFC for a complete guide to reply codes. Most FTP clients support a verbose mode which will allow the user to see these codes as commands progress.

If the FTP command requires the server to move a large piece of data (like a file), a second TCP connection is required to do this. This is referred to as the FTP data connection (as opposed to the aforementioned command connection). In active mode the data connection is opened by the server back to a listening client. In passive mode the client opens also the data connection. The data connection persists only for transporting the required data. It is closed as soon as all the data has been sent.

17.4.1 Active mode FTP

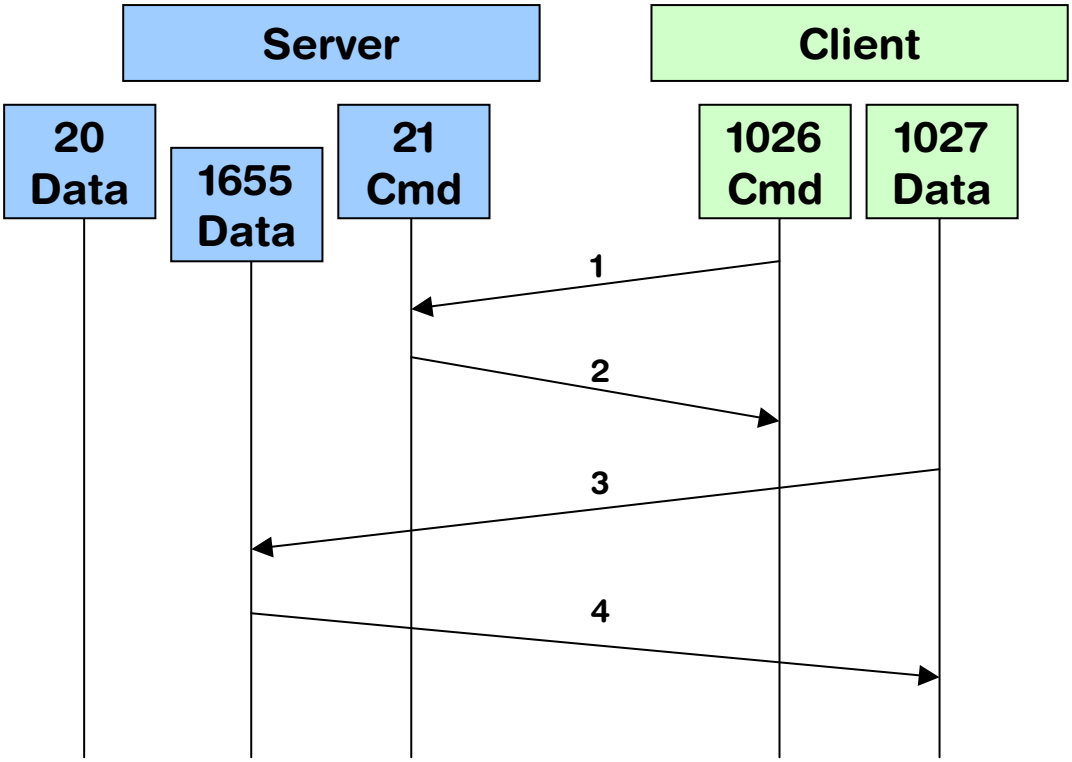
In active mode FTP the client connects from a random unprivileged port P (P > 1023) to the FTP server's command port, port 21. Then, the client starts listening to port P+1 and sends the FTP command PORT P+1 to the FTP server. The server will then connect back to the client's specified data port from its local data port, which is port 20.



17.4.2 Passive mode FTP

In passive mode FTP the client connects from a random unprivileged port P (P > 1023) to the FTP server's command port, port 21. In opposite to an active mode FTP connection where the client opens a passive port for data transmission and waits for the connection from server-side, the client sends in passive mode the "PASV" command to the server and expects an answer with the information on which port the server is listening for the data connection.

After receiving this information, the client connects to the specified data port of the server from its local data port.



17.4.3 FTP reply codes

Every FTP command is answered by one or more reply codes defined in *[RFC 959]*. A reply is an acknowledgment (positive or negative) sent from server to user via the control connection in response to FTP commands. The general form of a reply is a 3-digit completion code (including error codes) followed by Space <SP>, followed by one line of text and terminated by carriage return line feed <CRLF>. The codes are for use by programs and the text is usually intended for human users.

The first digit of the reply code defines the class of response. There are 5 values for the first digit:

- 1yz: Positive preliminary reply
- 2yz: Positive completion reply
- 3yz: Positive intermediate reply
- 4yz: Transient negative Completion reply
- 5yz: Permanent negative Completion reply

The second digit of the reply code defines the group of the response.

- x0z: Syntax - Syntax errors, syntactically correct commands that don't fit any functional category, unimplemented or superfluous commands.
- x1z: Information - These are replies to requests for information, such as status or help.
- x2z: Connections - Replies referring to the control and data connections.
- x3z: Authentication and accounting - Replies for the login process and accounting procedures.
- x4z: Unspecified as yet.
- x5z: File system - These replies indicate the status of the Server file system vis-a-vis the requested transfer or other file system action.

The third digit gives a finer gradation of meaning in each of the function categories, specified by the second digit.

17.4.4 Supported FTP commands

embOS/IP FTP server supports a subset of the defined FTP commands. Refer to *[RFC 959]* for a complete detailed description of the FTP commands. The following FTP commands are implemented:

FTP commands	Description
CDUP	Change to parent directory
CWD	Change working directory
DELE	Delete
LIST	List
MKD	Make directory
NLST	Name list
NOOP	No operation
PASS	Password
PASV	Passive
PORT	Data port
PWD	Print the current working directory
QUIT	Logout
RETR	Retrieve
RMD	Remove directory
RNFR	Renamr from
RNTO	Rename to
SIZE	Size of file
STOR	Store
SYST	System
TYPE	Transfer type
USER	User name
XCUP	Change to parent directory
XMKD	Make directory
XPWD	Print the current working directory
XRMD	Remove directory

Table 17.1: embOS/IP FTP commands

17.5 Using the FTP server sample

Ready to use examples for Microsoft Windows and embOS/IP are supplied. If you use another TCP/IP stack the sample `OS_IP_FTPServer.c` has to be adapted. The sample application opens a port which listens on port 21 until an incoming connection is detected. If a connection has been established `IP_FTPS_Process()` handles the client request in an extra task, so that the server is further listening on port 21. The example application requires a file system to make data files available. Refer to *File system abstraction layer* on page 456 and *File system abstraction layer* on page 456 for detailed information.

17.5.1 Using the Windows sample

If you have MS Visual C++ 6.00 or any later version available, you will be able to work with a Windows sample project using embOS/IP FTP server. If you do not have the Microsoft compiler, an precompiled executable of the FTP server is also supplied. The base directory of the Windows sample application is `C:\FTP\`.

Building the sample program

Open the workspace `Start_FTPServer.dsw` with MS Visual Studio (for example, double-clicking it). There is no further configuration necessary. You should be able to build the application without any error or warning message.

The server uses the IP address of the host PC on which it runs. Open a FTP client and connect by entering the IP address of the host (`127.0.0.1`) to connect to the FTP server. The server accepts anonymous logins. You can also login with the user name "`Admin`" and the password "`Secret`".

17.5.2 Running the FTP server example on target hardware

The embOS/IP FTP server sample application should always be the first step to check the proper function of the FTP server with your target hardware.

Add all source files located in the following directories (and their subdirectories) to your project and update the include path:

- `Application`
- `Config`
- `Inc`
- `IP`
- `IP\IP_FS\[NameOfUsedFileSystem]`

It is recommended that you keep the provided folder structure.

The sample application can be used on the most targets without the need for changing any of the configuration flags. The server processes two connections using the default configuration.

Note: Two connections mean that the target can handle up one target. A target requires always two connection, one for the command transfer and one for the data transfers. Every connection is handled in an separate task. Therefore, the FTP server uses up to three tasks in the default configuration. One task which listens on port 21 and accepts connections and two tasks to process the accepted connection. To modify the number of connections only the macro `MAX_CONNECTIONS` has to be modified.

17.6 Access control

The embOS/IP FTP server supports a fine-grained access permission scheme. Access permissions can be defined on user-basis for every directory and every file. The access permission of a directory or a file is a combination of the following attributes: visible, readable and writable. To control the access permission four callback functions have be implemented in the user application. The callback functions are defined in the structure `FTPS_ACCESS_CONTROL`. For detailed information about these structure, refer to *Structure FTPS_ACCESS_CONTROL* on page 362.

17.6.1 pfFindUser()

Description

Callback function which checks if the user is valid.

Prototype

```
int (*pfFindUser)   ( const char * sUser );
```

Return value

0 - UserID invalid or unknown
0 < - UserID, no password required
0 > - UserID, password required

Parameter

Parameter	Description
sUser	[IN] User name.

Table 17.2: pfFindUser() parameter list

Example

```
enum {
  USER_ID_ANONYMOUS = 1,
  USER_ID_ADMIN
};

/*********************************************************************
*
*       _FindUser
*
*  Function description
*    Callback function for user management.
*    Checks if user name is valid.
*
*  Return value
*    0    UserID invalid or unknown
*  > 0    UserID, no password required
*  < 0    - UserID, password required
*/
static int _FindUser (const char * sUser) {
  if (strcmp(sUser, "Admin") == 0) {
    return - USER_ID_ADMIN;
  }
  if (strcmp(sUser, "anonymous") == 0) {
    return USER_ID_ANONYMOUS;
  }
  return 0;
}
```


17.6.2 pfCheckPass()

Description

Callback function which checks if the password is valid.

Prototype

```
int (*pfCheckPass)  (       int    UserId,
                      const char * sPass );
```

Parameter

Parameter	Description
UserId	[IN] Id number
Pass	[IN] Password string.

Table 17.3: pfCheckPass() parameter list

Example

```
enum {
  USER_ID_ANONYMOUS = 1,
  USER_ID_ADMIN
};

/*********************************************************************
*
*       _CheckPass
*
*  Function description
*    Callback function for user management.
*    Checks user password.
*
*  Return value
*    0    UserID know, password valid
*    1    UserID unknown or password invalid
*/
static int _CheckPass (int UserId, const char * sPass) {
  if ((UserId == USER_ID_ADMIN) && (strcmp(sPass, "Secret") == 0)) {
    return 0;
  } else {
    return 1;
  }
}
```


17.6.3 pfGetDirInfo()

Description

Callback function which checks the permissions of the connected user for every directory.

Prototype

```
int (*pfGetDirInfo) (       int    UserId,
                      const char * sDirIn,
                            char * pDirOut,
                            int    SizeOfDirOut );
```

Parameter

Parameter	Description
UserId	[IN] Id number
sDirIn	[IN] Directory to check permission for
pDirOut	[OUT] Directory that can be accessed
SizeOfDirOut	[IN] Size of buffer addressed by pDirOut

Table 17.4: pfGetDirInfo() parameter list

Example

```
/* Excerpt from IP_FTPServer.h */
#define IP_FTPS_PERM_VISIBLE  (1 << 0)
#define IP_FTPS_PERM_READ     (1 << 1)
#define IP_FTPS_PERM_WRITE    (1 << 2)

/* Excerpt from OS_IP_FTPServer.c */
/*********************************************************************
*
*       _GetDirInfo
*
*  Function description
*    Callback function for permission management.
*    Checks directory permissions.
*
*  Return value
*    Returns a combination of the following:
*    IP_FTPS_PERM_VISIBLE    - Directory is visible as a directory entry
*    IP_FTPS_PERM_READ       - Directory can be read/entered
*    IP_FTPS_PERM_WRITE      - Directory can be written to
*
*  Parameters
*    UserId        - User ID returned by _FindUser()
*    sDirIn        - Full directory path and with trailing slash
*    sDirOut       - Reserved for future use
*    DirOutSize    - Reserved for future use
*
*  Notes
*    In this sample configuration anonymous user is allowed to do anything.
*    Samples for folder permissions show how to set permissions for different
*    folders and users. The sample configures permissions for the following
*    directories:
*      - /READONLY/: This directory is read only and can not be written to.
*      - /VISIBLE/ : This directory is visible from the folder it is located
*                    in but can not be accessed.
*      - /ADMIN/   : This directory can only be accessed by the user "Admin".
*/
static int _GetDirInfo(int UserId, const char * sDirIn, char * sDirOut, int DirOut-
Size) {
  int Perm;

  (void)sDirOut;
```



```
  (void)DirOutSize;

  Perm = IP_FTPS_PERM_VISIBLE | IP_FTPS_PERM_READ | IP_FTPS_PERM_WRITE;

  if (strcmp(sDirIn, "/READONLY/") == 0) {
    Perm = IP_FTPS_PERM_VISIBLE | IP_FTPS_PERM_READ;
  }
  if (strcmp(sDirIn, "/VISIBLE/") == 0) {
    Perm = IP_FTPS_PERM_VISIBLE;
  }
  if (strcmp(sDirIn, "/ADMIN/") == 0) {
    if (UserId != USER_ID_ADMIN) {
      return 0;  // Only Admin is allowed for this directory
    }
  }
  return Perm;
}
```


17.6.4 pfGetFileInfo()

Description

Callback function which checks the permissions of the connected user for every directory.

Prototype

```
int (*pfGetFileInfo) (       int    UserId,
                       const char * sFileIn,
                             char * pFileOut,
                             int    SizeOfFileOut );
```

Parameter

Parameter	Description
UserId	[IN] Id number
sFileIn	[IN] File to check permission for
pFileOut	[OUT] File that can be accessed
SizeOfFileOut	[IN] Size of buffer addressed by pFileOut

Table 17.5: pfGetFileInfo() parameter list

Additional information

Providing a function for file permissions is optional. If using permisisons on directory level is sufficient for your needs pfGetFileInfo may be declared NULL in the FTPS_ACCESS_CONTROL function table.

Example

```
/* Excerpt from IP_FTPServer.h */
#define IP_FTPS_PERM_VISIBLE  (1 << 0)
#define IP_FTPS_PERM_READ     (1 << 1)
#define IP_FTPS_PERM_WRITE    (1 << 2)

/* Excerpt from OS_IP_FTPServer.c */
/*********************************************************************
*
*       _GetFileInfo
*
*  Function description
*    Callback function for permission management.
*    Checks file permissions.
*
*  Return value
*    Returns a combination of the following:
*    IP_FTPS_PERM_VISIBLE    - File is visible as a file entry
*    IP_FTPS_PERM_READ       - File can be read
*    IP_FTPS_PERM_WRITE      - File can be written to
*
*  Parameters
*    UserId        - User ID returned by _FindUser()
*    sFileIn       - Full path to the file
*    sFileOut      - Reserved for future use
*    FileOutSize   - Reserved for future use
*
*  Notes
*    In this sample configuration all file accesses are allowed. File
*    permissions are checked against directory permissions. Therefore it
*    is not necessary to limit permissions on files that reside in a
*    directory that already limits access.
*    Setting permissions works the same as for _GetDirInfo() .
*/
static int _GetFileInfo(int UserId, const char * sFileIn, char * sFileOut, int File-
OutSize) {
```



```
  int Perm;

  (void)UserId;
  (void)sFileIn;
  (void)sFileOut;
  (void)FileOutSize;

  Perm = IP_FTPS_PERM_VISIBLE | IP_FTPS_PERM_READ | IP_FTPS_PERM_WRITE;

  return Perm;
}
```


17.7 Configuration

The embOS/IP FTP server can be used without changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which match the requirements of most applications.

The following types of configuration macros exist:

Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

Numerical values "N"

Numerical values are used somewhere in the source code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

Alias "A"

A macro which operates like a simple text substitute. An example would be the define `U8`, which the preprocessor would replace with `unsigned char`.

Function replacements "F"

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the source code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).

17.7.1 Compile time configuration switches

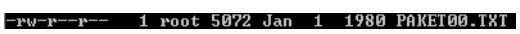
Type	Symbolic name	Default	Description
F	FTPS_WARN	--	Defines a function to output warnings. In debug configurations (DEBUG == 1) FTPS_WARN maps to IP_Warnf_Application()
F	FTPS_LOG	--	Defines a function to output logging messages. In debug configurations (DEBUG == 1) FTPS_LOG maps to IP_Logf_Application().
N	FTPS_AUTH_BUFFER_SIZE	32	Defines the size of the buffer used for the authentication string.
N	FTPS_BUFFER_SIZE	512	Defines the size of the send and receive buffer of the FTP server.
N	FTPS_MAX_PATH	128	Defines the maximum length of the buffer used for the path string.
N	FTPS_MAX_PATH_DIR	64	Defines the maximum length of the buffer used for the directory string.
N	FTPS_ERR_BUFFER_SIZE	256	Defines the size of the buffer used for the authentication string.

17.7.2 FTP server system time

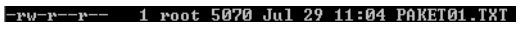
The FTP server requires a system time for the transmission of a complete file timestamp. FTP servers send only a piece of the timestamp of a file, either month, day and year or month, day and time. For the decision which pieces of the timestamp has to be transmitted, it compares the year of the current system time with the year which is stored in the timestamp of the file. Depending on the result of this comparison either the year or the time will be send. The following two examples show the output for both cases.

Example

1. If the value for year in the timestamp of the file is smaller then the value for year in the current system time, year will be sent:
   ```
   -rw-r--r--   1 root 2000 Jan 1  2007 PAKET00.TXT
   ```
   In this case, the FTP client leaves this column empty or fills the missing time with 00:00. The following screenshot shows the output of the Microsoft Windows command line FTP client:

2. If the value for year in the timestamp of the file is identical to the value for year in the current system time, the time (HH:MM) will be sent:
   ```
   -rw-r--r--   1 root 1000 Jul 29 11:04 PAKET01.TXT
   ```
   In this case, the FTP client leaves this column empty or fills the missing year with the current year. The following screenshot shows the output of the Microsoft Windows command line FTP client:

In the example, the value for the current time and date is defined to 1980-01-01 00:00. Therefore, the output will be similar to example 1., since no real time clock (RTC) has been implemented. Refer to *pfGetTimeDate()* on page 357 for detailed information.

17.7.2.1 pfGetTimeDate()

Description

Returns the current system time.

Prototype

```
int (*pfGetTimeDate) ( void );
```

Return value

Current system time. If no real time clock is implemented, it should return 0x00210000 (1980-01-01 00:00)

Additional information

The format of the time is arranged as follows:
Bit 0-4: 2-second count (0-29)
Bit 5-10: Minutes (0-59)
Bit 11-15: Hours (0-23)
Bit 16-20: Day of month (1-31)
Bit 21-24: Month of year (1-12)
Bit 25-31: Number of years since1980 (0-127)

This function pointer is used in the `FTPS_APPLICATION` structure. Refer to *Structure FTPS_APPLICATION* on page 363 for further information.

Example

```
static U32 _GetTimeDate(void) {
  U32 r;
  U16 Sec, Min, Hour;
  U16 Day, Month, Year;

  Sec   = 0;         // 0 based.  Valid range: 0..59
  Min   = 0;         // 0 based.  Valid range: 0..59
  Hour  = 0;         // 0 based.  Valid range: 0..23
  Day   = 1;         // 1 based.  Means that 1 is 1.
                     //           Valid range is 1..31 (depending on month)
  Month = 1;         // 1 based.  Means that January is 1. Valid range is 1..12.
  Year  = 28;        // 1980 based. Means that 2008 would be 28.
  r   = Sec / 2 + (Min << 5) + (Hour  << 11);
  r  |= (U32)(Day + (Month << 5) + (Year  << 9)) << 16;
  return r;
}
```


17.8 API functions

Function	Description
IP_FTPS_Process()	Initializes and starts the embOS/IP FTP server.
IP_FTPS_OnConnectionLimit()	Returns when the connection is closed or a fatal error occurs.

Table 17.6: embOS/IP FTP server interface function overview

17.8.1 IP_FTPS_Process()

Description

Initializes and starts the FTP server.

Prototype

```
int  IP_FTPS_Process ( const IP_FTPS_API      * pIP_API,
                       const void             * pConnectInfo,
                       const IP_FS_API        * pFS_API,
                       const FTPS_APPLICATION * pApplication );
```

Parameter

Parameter	Description
pIP_API	[IN] Pointer to a structure of type IP_FTPS_API.
pConnectInfo	[IN] Pointer to the connection info.
pFS_API	[IN] Pointer to the used file system API.
pApplication	[IN] Pointer to a structure of type FTPS_APPLICATION.

Table 17.7: IP_FTPS_Process() parameter list

Additional information

The structure type IP_FTPS_API contains mappings of the required socket functions to the actual IP stack. This is required because the socket functions are slightly different on different systems. The connection info is the socket which was created when the client has been connected to the command port (usually port 21). For detailed information about the structure type IP_FS_API refer to *Appendix A - File system abstraction layer* on page 455. For detailed information about the structure type FTPS_APPLICATION refer to *Structure FTPS_APPLICATION* on page 363.

17.8.2 IP_FTPS_OnConnectionLimit()

Description

Returns when the connection is closed or a fatal error occurs.

Prototype

```
void IP_FTPS_OnConnectionLimit( const IP_FTPS_API * pIP_API,
                                      void        * CtrlSock );
```

Parameter

Parameter	Description
pIP_API	[IN] Pointer to a structure of type IP_FTPS_API.
CtrlSock	[IN] Pointer to the socket which is related to the command connection.

Table 17.8: IP_FTPS_OnConnectionLimit() parameter list

Additional information

The structure type IP_FTPS_API contains mappings of the required socket functions to the actual IP stack. This is required because the socket functions are slightly different on different systems.

17.9 FTP server data structures

17.9.1 Structure IP_FTPS_API

Description

This structure contains the pointer to the socket functions which are required to use the FTP server.

Prototype

```
typedef struct {
  int (*pfSend) (const unsigned char * pData, int len, FTPS_SOCKET Socket);
  int (*pfReceive) (unsigned char * pData, int len, FTPS_SOCKET Socket);
  FTPS_SOCKET (*pfConnect)    (FTPS_SOCKET CtrlSocket, U16 Port);
  void        (*pfDisconnect) (FTPS_SOCKET DataSocket);
  FTPS_SOCKET (*pfListen) (FTPS_SOCKET CtrlSocket,U16 * pPort,U8 * pIPAddr);
  int (*pfAccept) (FTPS_SOCKET CtrlSocket, FTPS_SOCKET * pDataSocket);
} IP_FTPS_API;
```

Member	Description
pfSend	Callback function that sends data to the client on socket level.
pfReceive	Callback function that receives data from the client on socket level.
pfConnect	Callback function that handles the connect back to a FTP client on socket level if not using passive mode.
pfDisconnect	Callback function that disconnects a connection to the FTP client on socket level if not using passive mode.
pfListen	Callback function that binds the server to a port and addr.
pfAccept	Callback function that accepts incoming connections.

Table 17.9: Structure IP_FTPS_API member list

17.9.2 Structure FTPS_ACCESS_CONTROL

Description

This structure contains the pointer to the access control callback functions.

Prototype

```
typedef struct {
  int (*pfFindUser)    ( const char * sUser );
  int (*pfCheckPass)   (       int    UserId,
                         const char * sPass );
  int (*pfGetDirInfo) (        int    UserId,
                         const char * sDirIn,
                               char * sDirOut,
                               int    SizeOfDirOut );
} FTPS_ACCESS_CONTROL;
```

Member	Description
pfFindUser	Callback function that checks if the user is valid.
pfCheckPass	Callback function that checks if the password is valid.
pfGetDirInfo	Callback function that checks the permissions of the connected user for every directory.
pfGetFileInfo	Callback function that checks the permisisons of the connected user for every file. May be NULL if directory permisisons are sufficient for your needs.

Table 17.10: Structure FTPS_ACCESS_CONTROL member list

Example

Refer to *Access control* on page 349 for an example.

17.9.3 Structure FTPS_APPLICATION

Description

Used to store application specific parameters.

Prototype

```
typedef struct {
  FTPS_ACCESS_CONTROL * pAccess;
  U32 (*pfGetTimeDate) (void);
} FTPS_APPLICATION;
```

Member	Description
pAccess	Pointer to the FTPS_ACCESS_APPLCIATION structure.
pfGetTimeDate	Pointer to the function which returns the current system time.

Table 17.11: Structure FTPS_APPLICATION member list

Example

For additional information to structure FTPS_ACCESS_APPLICATION refer to *Structure FTPS_ACCESS_CONTROL* on page 362. For additional information to function pointer pfGetTimeDate() refer to *FTP server system time* on page 356.

Example

```
/* Excerpt from OS_IP_FTPServer.c */

/*********************************************************************
*
*       FTPS_ACCESS_CONTROL
*
*  Description
*   User/pass data table
*/
static FTPS_ACCESS_CONTROL _Access_Control = {
  _FindUser,
  _CheckPass,
  _GetDirInfo
};

*********************************************************************
*
*       _GetTimeDate
*/
static U32 _GetTimeDate(void) {
  U32 r;
  U16 Sec, Min, Hour;
  U16 Day, Month, Year;

  Sec   = 0;        // 0 based.  Valid range: 0..59
  Min   = 0;        // 0 based.  Valid range: 0..59
  Hour  = 0;        // 0 based.  Valid range: 0..23
  Day   = 1;        // 1 based.  Means that 1 is 1.
                    //           Valid range is 1..31 (depending on month)
  Month = 1;        // 1 based.  Means that January is 1. Valid range is 1..12.
  Year  = 28;       // 1980 based. Means that 2008 would be 28.
  r   = Sec / 2 + (Min << 5) + (Hour  << 11);
  r  |= (U32)(Day + (Month << 5) + (Year  << 9)) << 16;
  return r;
}

*********************************************************************
*
*       FTPS_APPLICATION
*
*  Description
*    Application data table, defines all application specifics
*    used by the FTP server
*/
static const FTPS_APPLICATION _Application = {
  &_Access_Control,
  _GetTimeDate
};
```


17.10 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the FTP server presented in the tables below have been measured on an ARM7 and a Cortrex-M3 system. Details about the further configuration can be found in the sections of the specific example.

Configuration used

```
#define FTPS_AUTH_BUFFER_SIZE   32
#define FTPS_BUFFER_SIZE       512
#define FTPS_MAX_PATH          128
#define FTPS_MAX_PATH_DIR       64
#define FTPS_ERR_BUFFER_SIZE  (FTPS_BUFFER_SIZE / 2)
```

17.10.1 ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP FTP server	approximately 6.6Kbyte

Table 17.12: FTP server ROM usage ARM7

17.10.2 ROM usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

Addon	ROM
embOS/IP FTP server	approximately 5.6Kbyte

Table 17.13: FTP server ROM usage Cortex-M3

17.10.3 RAM usage:

Almost all of the RAM used by the FTP server is taken from task stacks. The amount of RAM required for every child task depends on the configuration of your server. The table below shows typical RAM requirements for your task stacks.

Task	Description	RAM
ParentTask	Listens for incoming connections.	approximately 500 bytes
ChildTask	Handles a request.	approximately 1800 bytes

Table 17.14: FTP server RAM usage

Note: The FTP server requires at least 1 child task.

The approximately RAM usage for the FTP server can be calculated as follows:

RAM usage = 0.2 Kbytes + ParentTask + (NumberOfChildTasks * 1.8 Kbytes)

Example: FTP server accepting up only 1 connection

RAM usage = 0.2 Kbytes + 0.5 Kbytes + (1 * 1.8 Kbytes)
RAM usage = 2.5 Kbytes

Chapter 18

FTP client (Add-on)

The embOS/IP FTP client is an optional extension to the TCP/IP stack. The FTP client can be used with embOS/IP or with a different TCP/IP stack. All functions which are required to add a FTP client to your application are described in this chapter.

18.1 embOS/IP FTP client

The embOS/IP FTP client is an optional extension which adds the client part of FTP protocol to the stack. FTP stands for File Transfer Protocol. It is the basic mechanism for moving files between machines over TCP/IP based networks such as the Internet. FTP is a client/server protocol, meaning that one machine, the client, initiates a file transfer by contacting another machine, the server and making requests.

The FTP client implements the relevant parts of the following RFCs.

RFC#	Description
[RFC 959]	FTP - File Transfer Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc959.txt*

The following table shows the contents of the embOS/IP FTP client root directory:

Directory	Contents
`Application\`	Contains the example application to run the FTP client with embOS/IP.
`Config`	Contains the FTP client configuration file.
`Inc`	Contains the required include files.
`IP`	Contains the FTP client sources.
`IP\FS\`	Contains the sources for the file system abstraction layer and the read-only file system. Refer to *File system abstraction layer function table* on page 457 for detailed information.
`Windows\FTPclient\`	Contains the source, the project files and an executable to run embOS/IP FTP client on a Microsoft Windows host.

Supplied directory structure of embOS/IP FTP client package

18.2 Feature list

- Low memory footprint.
- Multiple connections supported.
- Independent of the file system: Any file system can be used.
- Independent of the TCP/IP stack: Any stack with sockets can be used.
- Demo application included.
- Project for executable on PC for Microsoft Visual Studio included.

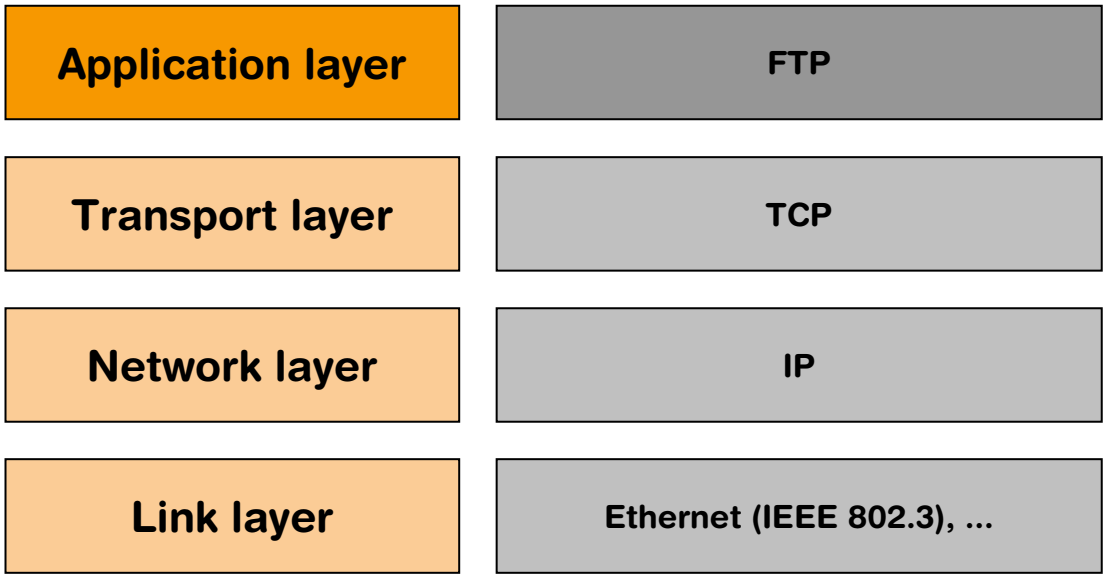
18.3 Requirements

TCP/IP stack

The embOS/IP FTP client requires a TCP/IP stack. It is optimized for embOS/IP, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of embOS/IP.

18.4 FTP basics

The File Transfer Protocol (FTP) is an application layer protocol. FTP is an unusual service in that it utilizes two ports, a 'Data' port and a 'CMD' (command) port. Traditionally these are port 21 for the command port and port 20 for the data port. FTP can be used in two modes, active and passive. Depending on the mode, the data port is not always on port 20.



When an FTP client contacts a server, a TCP connection is established between the two machines. The server does a passive open (a socket is listen) when it begins operation; thereafter clients can connect with the server via active opens. This TCP connection persists for as long as the client maintains a session with the server, (usually determined by a human user) and is used to convey commands from the client to the server, and the server replies back to the client. This connection is referred to as the FTP command connection.

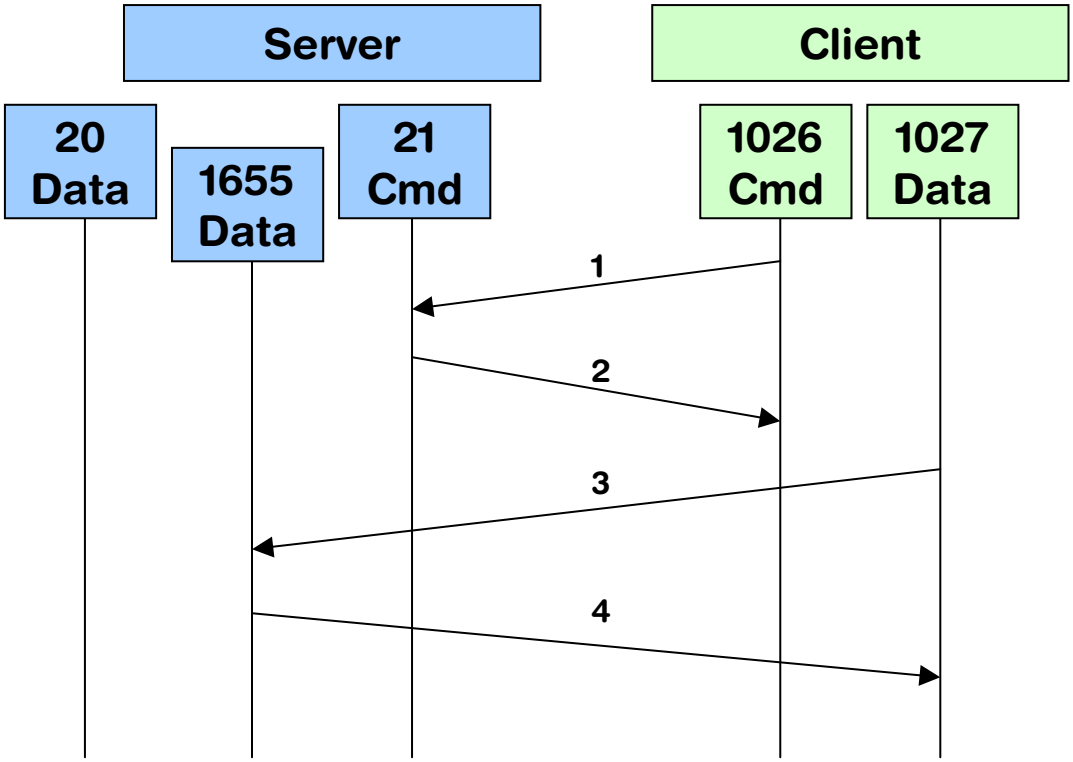
The FTP commands from the client to the server consist of short sets of ASCII characters, followed by optional command parameters. For example, the FTP command to display the current working directory is `PWD` (Print Working Directory). All commands are terminated by a carriage return-linefeed sequence (CRLF) (ASCII 10,13; or Ctrl-J, Ctrl-M). The servers replies consist of a 3 digit code (in ASCII) followed by some explanatory text. Generally codes in the 200s are success and 500s are failures. See the RFC for a complete guide to reply codes. Most FTP clients support a verbose mode which will allow the user to see these codes as commands progress.

If the FTP command requires the server to move a large piece of data (like a file), a second TCP connection is required to do this. This is referred to as the FTP data connection (as opposed to the aforementioned command connection). In active mode the data connection is opened by the server back to a listening client. In passive mode the client opens also the data connection. The data connection persists only for transporting the required data. It is closed as soon as all the data has been sent.

18.4.1 Passive mode FTP

In passive mode FTP the client connects from a random unprivileged port P (P > 1023) to the FTP server's command port, port 21. In opposite to an active mode FTP connection where the client opens a passive port for data transmission and waits for the connection from server-side, the client sends in passive mode the "PASV" command to the server and expects an answer with the information on which port the server is listening for the data connection.

After receiving this information, the client connects to the specified data port of the server from its local data port.



Note: In the current version of embOS/IP, the FTP client supports only passive mode FTP. Active mode FTP will be added in one of the coming versions.

18.4.2 Supported FTP client commands

embOS/IP FTP client supports a subset of the defined FTP commands. Refer to *[RFC 959]* for a complete detailed description of the FTP commands. The following FTP commands are implemented:

FTP commands	Description
CDUP	Change to parent directory
CWD	Change working directory
LIST	List directory
MKD	Make driectory
PASS	Password
PWD	Print the current working directory
RETR	Retrieve
RMD	Remove directory
STOR	Store
TYPE	Transfer type
USER	User name

Table 18.1: embOS/IP FTP client commands

18.5 Configuration

The embOS/IP FTP client can be used without changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which match the requirements of most applications.

The following types of configuration macros exist:

Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

Numerical values "N"

Numerical values are used somewhere in the source code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

Alias "A"

A macro which operates like a simple text substitute. An example would be the define `U8`, which the preprocessor would replace with `unsigned char`.

Function replacements "F"

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the source code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).

18.5.1 Compile time configuration switches

Type	Symbolic name	Default	Description
F	FTPC_WARN	--	Defines a function to output warnings. In debug configurations (DEBUG == 1) FTPC_WARN should be mapped to IP_Warnf_Application()
F	FTPC_LOG	--	Defines a function to output logging messages. In debug configurations (DEBUG == 1) FTPC_LOG should be mapped to IP_Logf_Application().
N	FTPC_BUFFER_SIZE	512	Defines the size of the in and the out buffer of the FTP client. This means that the client requires the defined number of bytes for each buffer. For example, FTPC_BUFFER_SIZE == 512 means 1024 bytes RAM requirement.
N	FTPC_CTRL_BUFFER_SIZE	256	Defines the maximum length of the buffer used for the control channel.
N	FTPC_SERVER_REPLY_BUFFER_SIZE	128	Defines the maximum length of the buffer used for the server reply strings. This buffer is only required and used in debug builds. In release builds the memory will not be allocated.

18.6 API functions

Function	Description
IP_FTPC_Connect()	Establishes a connection to a FTP server.
IP_FTPC_Disconnect()	Closes an established connection to a FTP server.
IP_FTPC_ExecCmd()	Sends a command to a FTP server.
IP_FTPC_Init()	Initializes the embOS/IP FTP client.

Table 18.2: embOS/IP FTP client interface function overview

18.6.1 IP_FTPC_Connect()

Description

Establishes a connection to a FTP server.

Prototype

```
int IP_FTPC_Connect ( IP_FTPC_CONTEXT * pContext,
                      const char *      sServer,
                      const char *      sUser,
                      const char *      sPass,
                      unsigned          PortCmd,
                      unsigned          Mode );
```

Parameter

Parameter	Description
pContext	[IN] Pointer to a structure of type IP_FTPC_CONTEXT.
sServer	[IN] Dot-decimal IP address of a FTP server, for example "192.168.1.55".
sUser	[IN] User name if required for the authentication. Can be NULL.
sPass	[IN] Password if required for the authentication. Can be NULL.
PortCmd	[IN] Port number of the port which is in listening mode on the FTP server. The well-known port for an FTP server that is waiting for connections is 21.
Mode	[IN] FTP transfer mode.

Table 18.3: IP_FTPC_Connect() parameter list

Valid values for parameter Mode	Description
FTPC_MODE_PASSIVE	Use passive mode FTP.

Return value

0 on success.
1 on error. Illegal parameter (pContext == NULL).
-1 on error during the process of establishing a connection.

Additional information

The function IP_FTPC_Init() must be called before a call IP_FTPC_Connect(). For detailed information about IP_FTPC_Init() refer to *IP_FTPC_Init()* on page 380.

Note: In the current version of embOS/IP, the FTP client supports only passive mode FTP.

Example

Refer to *IP_FTPC_ExecCmd()* on page 376 for an example application which uses IP_FTPC_Connect().

18.6.2 IP_FTPC_Disconnect()

Description

Closes an established connection to a FTP server.

Prototype

```
int IP_FTPC_Disconnect ( IP_FTPC_CONTEXT * pContext );
```

Parameter

Parameter	Description
pContext	[IN] Pointer to a structure of type IP_FTPC_CONTEXT.

Table 18.4: IP_FTPC_Disconnect() parameter list

Return value

0 on success.
1 on error. Illegal parameter (pContext == NULL).

Example

Refer to *IP_FTPC_ExecCmd()* on page 376 for an example application which uses IP_FTPC_Disconnect().

18.6.3 IP_FTPC_ExecCmd()

Description

Executes a command on the FTP server.

Prototype

```
int IP_FTPC_ExecCmd ( IP_FTPC_CONTEXT * pContext,
                      unsigned Cmd,
                      const char * sPara );
```


Parameter

Parameter	Description
pContext	[IN] Pointer to a structure of type IP_FTPC_CONTEXT.
Cmd	[IN] See table below.
sPara	[IN] String with the required parameters for the command. Depending on the command, the parameter can be NULL.

Table 18.5: IP_FTPC_ExecCmd() parameter list

Valid values for parameter Cmd	Description
FTPC_CMD_CDUP	The command CDUP (Change to Parent Directory). sPara is NULL.
FTPC_CMD_CWD	The command CWD (Change Working Directory). sPara is the path to the directory that should be accessed.
FTPC_CMD_LIST	The command LIST (List current directory content). sPara is NULL.
FTPC_CMD_MKD	The command MKD (Make directory). sPara is the name of the directory that should be created.
FTPC_CMD_PASS	The command PASS (Set password). sPara is the password.
FTPC_CMD_PWD	The command PWD (Print Working Directory). sPara is NULL.
FTPC_CMD_RETR	The command RETR (Retrieve). sPara is the name of the file that should be received from the server. The FTP client creates a file on the used storage medium and stores the retrieved file.
FTPC_CMD_RMD	The command RMD (Remove directory). sPara is the name of the directory that should be removed.
FTPC_CMD_STOR	The command STOR (Store). sPara is the name of the file that should be stored on the server. The FTP client opens the file and transmits it to the FTP server.
FTPC_CMD_TYPE	The command TYPE (Transfer type). sPara is the transfer type.
FTPC_CMD_USER	The command USER (Set username). sPara is the username.

Return value

0 on success.
1 on error. Illegal parameter (pContext == NULL).
-1 on error during command execution.

Additional information

IP_FTPC_Init() and IP_FTPC_Connect() have to be called before IP_FTPC_ExecCmd(). Refer to *IP_FTPC_Init()* on page 380 for detailed information about how to initialize the FTP client and refer to *IP_FTPC_Connect()* on page 374 for detailed information about how to establish a connection to a FTP server.

`IP_FTPC_ExecCmd()` sends a command to the server and handles everything what is required on FTP client side. The commands which are listed in section *Supported FTP client commands* on page 371, but not explained here, are normally not directly called from the user application. There is no need to call IP_ExecCmd() with these commands. The FTP client uses these commands internally and sends them to the server if required. For example, the call of `IP_FTPC_Connect()` sends the the commands USER, PASS and SYST to the server and process the server replies for each of the commands, an explicit call of IP_FTPC_Exec() with one of these commands is not requried.

Example

```
/* Excerpt from the example application OS_IP_FTPClient.c */

/*********************************************************************
*
*       MainTask
*
*  Note:
*   The size of the stack of this task should be at least
*   1200 bytes + FTPC_CTRL_BUFFER_SIZE + 2 * FTPC_BUFFER_SIZE.
*/
void MainTask(void);
void MainTask(void) {
  IP_FTPC_CONTEXT FTPConnection;
  U8 acCtrlIn[FTPC_CTRL_BUFFER_SIZE];
  U8 acDataIn[FTPC_BUFFER_SIZE];
  U8 acDataOut[FTPC_BUFFER_SIZE];
  int r;

  //
  // Initialize the IP stack
  //
  IP_Init();
  OS_CREATETASK(&_TCB, "IP_Task", IP_Task  , 150, _IPStack);  // Start the IP_Task
  //
  // Check if target is configured
  //
  while (IP_IFaceIsReady() == 0) {
    BSP_ToggleLED(1);
    OS_Delay(50);
  }
  //
  // FTP client task
  //
  while (1) {
    BSP_SetLED(0);
    //
    // Initialize FTP client context
    //
    memset(&FTPConnection, 0, sizeof(FTPConnection));
    //
    // Initialize the FTP client
    //
    IP_FTPC_Init(&FTPConnection, &_IP_Api, &IP_FS_FS, acCtrlIn, sizeof(acCtrlIn),
                 acDataIn, sizeof(acDataIn), acDataOut, sizeof(acDataOut));
    //
    // Connect to the FTP server
    //
    r = IP_FTPC_Connect(&FTPConnection, "192.168.199.164", "Admin", "Secret",
                        21, FTPC_MODE_PASSIVE);
    if (r == FTPC_ERROR) {
      FTPC_LOG(("APP: Could not connect to FTP server.\r\n"));
      goto Disconnect;
    }
    //
    // Change from root directory into directory "Test"
```



```
    //
    r = IP_FTPC_ExecCmd(&FTPConnection, FTPC_CMD_CWD,  "/Test/");
    if (r == FTPC_ERROR) {
      FTPC_LOG(("APP: Could not change working directory.\r\n"));
      goto Disconnect;
    }
    //
    // Upload the file "Readme.txt
    //
    r = IP_FTPC_ExecCmd(&FTPConnection, FTPC_CMD_STOR, "Readme.txt");
    if (r == FTPC_ERROR) {
      FTPC_LOG(("APP: Could not upload data file.\r\n"));
      goto Disconnect;
    }
    //
    // Change back to root directory.
    //
    r = IP_FTPC_ExecCmd(&FTPConnection, FTPC_CMD_CDUP, NULL);
    if (r == FTPC_ERROR) {
      FTPC_LOG(("APP: Change to parent directory failed.\r\n"));
      goto Disconnect;
    }
    //
    // Disconnect.
    //
Disconnect:
    IP_FTPC_Disconnect(&FTPConnection);
    FTPC_LOG(("APP: Done.\r\n"));
    BSP_ClrLED(0);
    OS_Delay (10000);
  }
}
```


18.6.4 IP_FTPC_Init()

Description

Initializes the FTP client context.

Prototype

```
int IP_FTPC_Init ( IP_FTPC_CONTEXT *   pContext,
                   const IP_FTPC_API * pIP_API,
                   const IP_FS_API *   pFS_API,
                   U8 *                pCtrlBuffer,
                   unsigned            NumBytesCtrl,
                   U8 *                pDataInBuffer,
                   unsigned            NumBytesDataIn,
                   U8 *                pDataOutBuffer,
                   unsigned            NumBytesDataOut );
```

Parameter

Parameter	Description
pContext	[IN] Pointer to a structure of type IP_FTPC_CONTEXT.
pIP_API	[IN] Pointer to a structure of type IP_FTPC_API.
pFS_API	[IN] Pointer to the file system API.
pControlBuffer	[IN] Pointer to the buffer used for the control channel information.
NumBytesCtrl	[IN] Size of the control buffer in bytes.
pDataInBuffer	[IN] Pointer to the buffer used to receive data from the server.
NumBytesDataIn	[IN] Size of the receive buffer in bytes.
pDataOutBuffer	[IN] Pointer to the buffer used to transmit data to the server.
NumBytesDataOut	[IN] Size of the transmit buffer in bytes.

Table 18.6: IP_FTPC_Init() parameter list

Return value

0 on success.
1 on error. Invalid parameters.

Additional information

IP_FTPC_Init() must be called before any other FTP client function will be called. For detailed information about the structure type IP_FS_API refer to *Appendix A - File system abstraction layer* on page 455. For detailed information about the structure type IP_FTPC_API refer to *Appendix A - File system abstraction layer* on page 455.

Example

Refer to *IP_FTPC_ExecCmd()* on page 376 for an example application which uses IP_FTPC_Init().

18.7 FTP client data structures

18.7.1 Structure IP_FTPC_API

Description

This structure contains the pointer to the socket functions which are required to use the FTP client.

Prototype

```
typedef struct {
  FTPC_SOCKET  (*pfConnect)    (const char * SrvAddr, unsigned SrvPort);
  void         (*pfDisconnect) (FTPC_SOCKET Socket);
  int          (*pfSend)       (const char * pData, int Len,
                                FTPC_SOCKET Socket);
  int          (*pfReceive)    (char * pData, int Len, FTPC_SOCKET Socket);
} IP_FTPC_API;
```

Member	Description
pfConnect	Callback function that handles the connect to a FTP server on socket level.
pfDisconnect	Callback function that disconnects a connection to the FTP server on socket level.
pfSend	Callback function that sends data to the FTP server on socket level.
pfReceive	Callback function that receives data from the FTP server on socket level.

Table 18.7: Structure IP_FTPC_API member list

18.8 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the FTP client presented in the tables below have been measured on an ARM7 and a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

Configuration used

```
#define FTPC_BUFFER_SIZE               512
#define FTPC_CTRL_BUFFER_SIZE          256
#define FTPC_SERVER_REPLY_BUFFER_SIZE  128  // Only required in debug builds
                                            // with enabled logging.
```

18.8.1 ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP FTP client	approximately 2Kbyte

Table 18.8: FTP client ROM usage ARM7

18.8.2 ROM usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

Addon	ROM
embOS/IP FTP client	approximately 1.7Kbyte

Table 18.9: FTP client ROM usage Cortex-M3

18.8.3 RAM usage:

Almost all of the RAM used by the web server is taken from task stacks. The amount of RAM required for every child task depends on the configuration of your client. The table below shows typical RAM requirements for your task stacks.

Build	Description	RAM
Release	A task used for the FTP client without debugging features and disabled debug outputs.	app. 1400 bytes

Table 18.10: FTP client RAM usage release build

The approximately task stack size required for the FTP client can be calculated as follows:

TaskStackSize = 2 * FTPC_BUFFER_SIZE + FTPC_CTRL_BUFFER_SIZE

Build	Description	RAM
Debug	A task used for the FTP client with debugging features and enabled debug outputs.	app. 1550 bytes

Table 18.11: FTP client RAM usage debug build

The approximately task stack size required for the FTP client can be calculated as follows:

TaskStackSize = 2 * FTPC_BUFFER_SIZE + FTPC_CTRL_BUFFER_SIZE
+ FTPC_SERVER_REPLY_BUFFER_SIZE

Chapter 19

PPP / PPPoE (Add-on)

The embOS/IP implementation of the Point to Point Protocol (PPP) is an optional extension to embOS/IP. It can be used to establish a PPP connection over Ethernet (PPPoE) or using modem to connect via telephone carrier. All functions that are required to add PPP/PPPoE to your application are described in this chapter.

19.1 embOS/IP PPP/PPPoE

The embOS/IP PPP implementation is an optional extension which can be seamlessly integrated into your TCP/IP application. It combines a maximum of performance with a small memory footprint. The PPP implementation allows an embedded system to connect via Point to Point Protocol to a network.

The PPP module implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 1334]	PPP Authentication Protocols Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1334.txt*
[RFC 1661]	The Point-to-Point Protocol (PPP) Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1661.txt*
[RFC 1994]	PPP Challenge Handshake Authentication Protocol (CHAP) Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1994.txt*
[RFC 2516]	A Method for Transmitting PPP Over Ethernet (PPPoE) *Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc2516.txt*

The following table shows the contents of the embOS/IP root directory:

Directory	Content
`Application`	Contains the example application to run the PPP implementation with embOS/IP.
`Inc`	Contains the required include files.
`IP`	Contains the PPP sources, `IP_PPP.c`, `IP_PPP_CCP.c`, `IP_PPP_Int.h`, `IP_PPP_IPCP.c`, `IP_PPP_LCP.c`, IP_PPP_Line.c, IP_PPP_PAP.c `and` `IP_PPPoE.c`. Additionally to the main source code files of the PPP add-on an example implementation for the connection of a modem via USART (IP_Modem_UART.c) is supplied.

Supplied directory structure of embOS/IP PPP package

19.2 Feature list

- Low memory footprint.
- Support PAP authentication protocol
- Support for PPP over Ethernet.

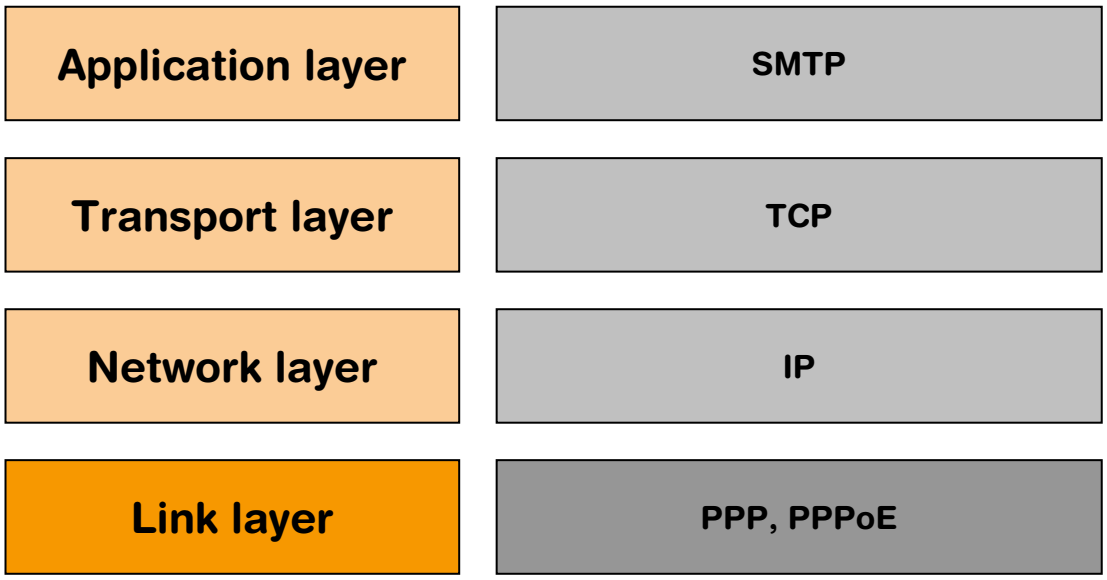
19.3 Requirements

TCP/IP stack

The embOS/IP PPP implementation requires the embOS/IP TCP/IP stack. Your modem has to be able to be configured to respond in the format "<CR><LF><Response>".

19.4 PPP backgrounds

The Point to Point Protocol is a link layer protocol for establishing a direct connection between two network nodes.



Using PPP, an embOS/IP application can establish a PPP connection to a PPP server. The handshaking mechanism includes normally an authentication process. The current version of embOS/IP supports the the following authentication schemes:

- PAP - Password Authentication Protocol

19.5 API functions

Function	Description
PPPoE functions	
IP_PPPOE_AddInterface()	Adds a PPPoE interface.
IP_PPPOE_ConfigRetries()	Configures the number of times to resend a lost message before breaking the connection.
IP_PPPOE_Reset()	Resets the PPPoE connection.
IP_PPPOE_SetAuthInfo()	Sets the authentication information for the PPPoE connection.
IP_PPPOE_SetUserCallback()	Sets a callback function to inform the user about a status change of the connection.
PPP functions	
IP_PPP_AddInterface()	Adds a PPP driver.
IP_PPP_OnRx()	Receives one or more characters from the hardware.
IP_PPP_OnRxChar()	Receives a character from the hardware.
IP_PPP_OnTxChar()	Sends a character via PPP.
IP_PPP_SetUserCallback()	Sets a callback function to inform the user about a status change of the connection.
Modem functions	
IP_MODEM_Connect()	Connects using the modem line.
IP_MODEM_Disconnect()	Disconnects the modem line.
IP_MODEM_GetResponse()	Retrieves the last received responses from the modem.
IP_MODEM_SendString()	Sends a command to the modem.
IP_MODEM_SendStringEx()	Sends a command to the modem and checks for the correct response.
IP_MODEM_SetAuthInfo()	Sets authentication information required by your ISP.
IP_MODEM_SetConnectTimeout()	Sets the timeout how long to wait until the modem is fully connected.
IP_MODEM_SetInitCallback()	Sets a callback proving modem initializations.
IP_MODEM_SetInitString()	Sets a single command needed for modem initialization.
IP_MODEM_SetSwitchToCmdDelay()	Sets a delay when sending "+++ATH" is problematic.

Table 19.1: embOS/IP PPP/PPPoE/Modem API function overview

19.6 PPPoE functions

19.6.1 IP_PPPOE_AddInterface()

Description

Adds a PPPoE interface.

Prototype

```
int IP_PPPOE_AddInterface( unsigned HWIFace );
```

Parameter

Parameter	Description
HWIFace	[IN] Zero-based index of available network interfaces.

Table 19.2: IP_PPPOE_AddInterface() parameter list

Return value

>= 0 Index of the interface.

19.6.2 IP_PPPOE_ConfigRetries()

Description

Configures the number of times to resend a lost message before breaking the connection.

Prototype

```
void IP_PPPOE_ConfigRetries( unsigned HWIFace,
                             U32      NumTries,
                             U32      Timeout );
```

Parameter

Parameter	Description
HWIFace	[IN] Zero-based index of available network interfaces.
NumTries	[IN] Number of times the stack will resend the message.
Timeout	[IN] Timout in ms before a resend is triggered.

Table 19.3: IP_PPPOE_ConfigRetries() parameter list

19.6.3 IP_PPPOE_Reset()

Description

Resets the PPPoE connection. The PPPoE layer is closed by sending a PADT if connected. Also resets the PPP connection state, but does not send any more PPP packets.

Prototype

```
void IP_PPPOE_Reset( unsigned HWIFace );
```

Parameter

Parameter	Description
HWIFace	[IN] Zero-based index of available network interfaces.

Table 19.4: IP_PPPOE_Reset() parameter list

19.6.4 IP_PPPOE_SetAuthInfo()

Description

Sets the authentication information for the PPPoE connection.

Prototype

```
void IP_PPPOE_SetAuthInfo(unsigned     IFaceId,
                          const char * sUser,
                          const char * sPass );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.
sUser	[IN] PPPoE user name.
sPass	[IN] PPPoE user password.

Table 19.5: IP_PPPOE_SetAuthInfo() parameter list

19.6.5 IP_PPPOE_SetUserCallback()

Description

Sets a callback function to inform the user about a status change.

Prototype

```
void IP_PPPOE_SetUserCallback( U32                          IFaceId,
                               IP_PPPOE_INFORM_USER_FUNC * pfInformUser );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.
pfInformUser	[IN] Pointer to a user function of type IP_PPPOE_INFORM_USER_FUNC which is called when a status change occurs.

Table 19.6: IP_PPP_SetUserCallback() parameter list

Additional Information

Callback function will only be added if IP_PPPOE_AddInterface() has been called before.

IP_PPPOE_INFORM_USER_FUNC is defined as follows:

```
typedef void (IP_PPPOE_INFORM_USER_FUNC)(U32 IFaceId, U32 Status);
```


19.7 PPP functions

19.7.1 IP_PPP_AddInterface()

Description

Adds a PPP driver.

Prototype

```
int  IP_PPP_AddInterface( const IP_PPP_LINE_DRIVER * pLineDriver,
                          int                        ModemIndex);
```

Parameter

Parameter	Description
pLineDriver	[IN] Pointer to a Structure IP_PPP_LINE_DRIVER.
ModemIndex	[IN] Modem index; Fixed to 0.

Table 19.7: IP_PPP_AddInterface() parameter list

Return value

>= 0 Index of the interface.

19.7.2 IP_PPP_OnRx()

Description

Receives one or more characters from the hardware. Uses IP_PPP_OnRxChar() to receive the characters one by one.

Prototype

```
void IP_PPP_OnRx( struct IP_PPP_CONTEXT * pContext,
                  U8                    * pData,
                  int                     NumBytes );
```

Parameter

Parameter	Description
pContext	[IN] Pointer to a Structure IP_PPP_CONTEXT.
pData	[IN] Pointer to a buffer which is storing the received data.
NumBytes	[IN] Number of received bytes.

Table 19.8: IP_PPP_OnRx() parameter list

19.7.3 IP_PPP_OnRxChar()

Description

Receives a character from the hardware. Checks if the received character is an escape character, removes the escape character if equired and stores the character into packet buffer. When a complete packet is received, it is given to the stack.

Prototype

```
void IP_PPP_OnRxChar( struct IP_PPP_CONTEXT * pContext,
                      U8                      Data );
```

Parameter

Parameter	Description
pContext	[IN] Pointer to a Structure IP_PPP_CONTEXT.
Data	[IN] 1 character.

Table 19.9: IP_PPP_OnRxChar() parameter list

19.7.4 IP_PPP_OnTxChar()

Description

Sends a character via PPP. The function checks if the character needs an escape character for the HDLC framing and sends the the escape character if required.

Prototype

```
void IP_PPP_OnTxChar( unsigned Unit );
```

Parameter

Parameter	Description
Unit	Typically 0.

Table 19.10: IP_PPP_OnTxChar() parameter list

19.7.5 IP_PPP_SetUserCallback()

Description

Sets a callback function to inform the user about a status change.

Prototype

```
void IP_PPP_SetUserCallback( U32                       IFaceId,
                             IP_PPP_INFORM_USER_FUNC * pfInformUser );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.
pfInformUser	[IN] Pointer to a user function of type IP_PPP_INFORM_USER_FUNC which is called when a status change occurs.

Table 19.11: IP_PPP_SetUserCallback() parameter list

Additional Information

Callback function will only be added if IP_PPP_AddInterface() has been called before.

IP_PPP_INFORM_USER_FUNC is defined as follows:

```
typedef void (IP_PPP_INFORM_USER_FUNC)(U32 IFaceId, U32 Status);
```


19.8 Modem functions

19.8.1 IP_MODEM_Connect()

Description

Initializes a PPP connect on a modem using the passed AT command.

Prototype

```
int IP_MODEM_Connect( const char * sATCommand );
```

Parameter

Parameter	Description
sATCommand	[IN] AT command string to dial up a connection. Must not use <CR> at the end of the dial string. Typically this is the command “ATD” followed by a number to dial.

Table 19.12: IP_MODEM_Connect() parameter list

Return value

0: Connected
!= 0: Error

Example

```
IP_MODEM_Connect(“ATD*99***1#“);
```


19.8.2 IP_MODEM_Disconnect()

Description

Disconnects the connection established with a modem on a specific interface.

Prototype

```
void IP_MODEM_Disconnect( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.

Table 19.13: IP_MODEM_Disconnect() parameter list

Example

```
IP_MODEM_Disconnect(0);
```


19.8.3 IP_MODEM_GetResponse()

Description

Retrieves a pointer to the responses received since the last AT command sent.

Prototype

```
const char * IP_MODEM_GetResponse( unsigned   IFaceId,
                                   char     * pBuffer
                                   unsigned   NumBytes
                                   unsigned * pNumBytesInBuffer );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
pBuffer	[OUT] Pointer to the receive buffer where the response will be copied to. May be NULL.
NumBytes	[IN] Size of the buffer pointed to by pBuffer.
pNumBytesBuffer	[OUT] Number of bytes copied to pBuffer. May be NULL.

Table 19.14: IP_MODEM_GetResponse() parameter list

Return value

Pointer to the last responses received in the original buffer.

Example

```
U8 aBuffer[256];
unsigned NumBytesReceived;

IP_MODEM_SendString(0, "AT");
IP_MODEM_GetResponse(0, &aBuffer[0], sizeof(aBuffer), &NumBytesReceived);
```


19.8.4 IP_MODEM_SendString()

Description

Sends an AT command to the modem without waiting for an answer.

Prototype

```
void IP_MODEM_SendString( unsigned     IFaceId,
                          const char * sCmd );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
sCmd	[IN] AT command to be sent.

Table 19.15: IP_MODEM_SendString() parameter list

Example

```
IP_MODEM_SendString(0, "AT");
```

Additional information

Sending a new command with `IP_MODEM_SendString()` clears the buffer of previous received responses.

19.8.5 IP_MODEM_SendStringEx()

Description

Sends an AT command to the modem and waits for the expected response with a timeout or checks for responses received in multpiple parts.

Prototype

```
int IP_MODEM_SendStringEx( unsigned     IFaceId,
                           const char * sCmd,
                           const char * sResponse,
                           unsigned     Timeout,
                           unsigned     RecvBufOffs );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
sCmd	[IN] AT command to be sent. May be NULL.
sResponse	[IN] Expected response without <CR><LF> in front. May be NULL.
Timeout	[IN] Timeout to wait for any response in ms.
RecvBufOffs	[IN] Can be used to check for a response that is sent in multiple parts. See below for additional information. May be NULL.

Table 19.16: IP_MODEM_SendStringEx() parameter list

Return value

0: O.K., corect response received
1: Timeout
2: Wrong response received, check with IP_MODEM_GetResponse()

Additional information

Sending a new command with IP_MODEM_SendString() clears the buffer of previous received responses.

RecvBufOffs can be used to check for responses that are sent by the modem in multiple responses. If not passed '0' the receive buffer will not be cleared to not clear out already received following responses from the previously sent command. RecvBufOffs id the offset in bytes from the beginning of the first received response. Being able to receive responses that are sent in multiple parts is necessary as some command may be responded with a confirm for the command sent itself and respond with a second message after an undefined time.

Example sending a command and checking for its response with a timeout

```
IP_MODEM_SendStringEx(0, "AT", "OK", 100, 0);
```

Example for checking the SIM status of a GSM modem

```
int r;

//
// Check if the modem is waiting for a SIM PIN to be entered
//
r = IP_MODEM_SendStringEx(0, "AT+CPIN?\r", "+CPIN: SIM PIN", 1000, 0);
if (r == 0) {
  //
  // The modem is waiting for the PIN to be entered
  //
  IP_MODEM_SendString(0, "AT^SSET=1\r");  // Enable "^SSIM READY" response once
                                          // the SIM data has been read
  IP_OS_Delay(100);
```



```
  //
  // Enter SIM PIN. The OK response will arrive quickly. The modem then
  // reads data from the SIM.
  //
  IP_MODEM_SendStringEx(0, "AT+CPIN="1234"\r", "OK", 15000, 0);
  //
  // After receiving the “OK” response for the command the modem will need an
  // undefined time to read data from the SIM. The modem sends the response
  // “^SSIM READY“ once it has finished. We will receive the response at an
  // 6 byte offset (OK<CR><LF><CR><LF>^SSIM READY).
  //
  IP_MODEM_SendStringEx(0, NULL, "^SSIM READY", 15000, 6);
} else {
  //
  // The modem does not seem to wait for a PIN, check if the modem
  // reports "READY". This means no PIN is set for the SIM card. In this case
  // the modem responds with "+CPIN: READY" that will be located at offset 0
  // in the recive buffer.
  //
  if (IP_MEMCMP(IP_MODEM_GetResponse(0, NULL, 0, NULL), "+CPIN: READY", 12) != 0) {
    IP_Panic("Unrecognized response from modem.");
  }
}
```


19.8.6 IP_MODEM_SetAuthInfo()

Description

Sets authentication information if needed for the connection to establish.

Prototype

```
void IP_MODEM_SetAuthInfo( unsigned     IFaceId,
                           const char * sUser,
                           const char * sPass );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
sUser	[IN] String containing the user name to be used.
sPass	[IN] String containing the password to be used.

Table 19.17: IP_MODEM_SetAuthInfo() parameter list

Example

```
IP_MODEM_SetAuthInfo(0, "User", "Pass");
```

Additional information

Setting a user name and a password is only necessary when required by your ISP.

19.8.7 IP_MODEM_SetConnectTimeout()

Description

Sets the connect timeout to wait for a requested connection with IP_MODEM_Connect() to be established.

Prototype

```
void IP_MODEM_SetConnectTimeout( unsigned IFaceId,
                                 unsigned ms );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
ms	[IN] Timeout in ms. Default: 15s.

Table 19.18: IP_MODEM_SetConnectTimeout() parameter list

Example

```
IP_MODEM_SetConnectTimeout(0, 30000);
```


19.8.8 IP_MODEM_SetInitCallback()

Description

Sets a callback that is used to initialize the modem before actually starting the connection attempt. The callback is called from IP_MODEM_Connect().

Prototype

```
void IP_MODEM_SetInitCallback( void (*pfInit)(void) );
```

Parameter

Parameter	Description
pfInit	[IN] Void callback routine for intialization of the modem before connecting.

Table 19.19: IP_MODEM_SetInitCallback() parameter list

Example

```
static void _InitModem(void) {
  IP_MODEM_SendString(0, "AT");
}

IP_MODEM_SetInitCallback(_InitModem);
IP_MODEM_Connect("ATD*99***1#");
```


19.8.9 IP_MODEM_SetInitString()

Description

Sets an initialization string that is sent to the modem before actually starting the connection attempt. In case IP_MODEM_SetInitCallback() is used the init string is not sent.

Prototype

```
void IP_MODEM_SetInitString( const char * sInit );
```

Parameter

Parameter	Description
sInit	[IN] Command to be sent to the modem before connecting.

Table 19.20: IP_MODEM_SetInitString() parameter list

Example

```
IP_MODEM_SetInitString("ATE0V1");
IP_MODEM_Connect("ATD*99***1#");
```


19.8.10 IP_MODEM_SetSwitchToCmdDelay()

Description

Sets a delay that will be executed with "+++ATH" command when using IP_MODEM_Disconnect().

Prototype

```
void IP_MODEM_SetSwitchToCmdDelay( unsigned IFaceId,
                                   unsigned ms );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
ms	[IN] Timeout in ms between sending "+++" and "ATH".

Table 19.21: IP_MODEM_SetSwitchToCmdDelay() parameter list

Additional information

Sending "+++ATH" to switch back to command mode and then hanging up the connection is fine to be sent in one in one message. For some modem this does not apply. They need some time to switch back to command mode before accepting "ATH" for hanging up.

19.9 PPP data structures

Function	Description
IP_PPP_CONTEXT	Structure which stores the information about the PPP connection.
RESEND_INFO	A structure which stores the resend condition for different stages of the PPP connection.
IP_PPP_LINE_DRIVER	Structure with pointers to application related functions.

Table 19.22: embOS/IP PPP data structure overview

19.9.1 Structure IP_PPP_CONTEXT

Description

A structure which stores the information about the PPP connection.

Prototype

```
typedef struct IP_PPP_CONTEXT {
  PPP_SEND_FUNC * pfSend;
  PPP_TERM_FUNC * pfTerm;
  PPP_INFORM_USER_FUNC * pfInformUser;
  void          * pSendContext;
  int             NumBytesPrepend;
  U8              IFaceId;
  struct {
    U32            NumTries;
    I32            Timeout;
  } Config;
  struct {
    U8             Id;
    U8             aOptCnt[MAX_OPT];
    PPP_LCP_STATE  AState;
    PPP_LCP_STATE  PState;
    RESEND_INFO    Resend;
    U16            MRU;
    U32            ACCM;
    U32            OptMask;
  } LCP;
  struct {
    U8             Id;
    U8             aOptCnt[MAX_OPT];
    PPP_CCP_STATE  AState;
    PPP_CCP_STATE  PState;
    RESEND_INFO    Resend;
    U32            OptMask;
  } CCP;
  struct {
    U8             Id;
    U8             aOptCnt[MAX_OPT];
    PPP_IPCP_STATE AState;
    PPP_IPCP_STATE PState;
    RESEND_INFO    Resend;
    IP_ADDR        IpAddr;
    IP_ADDR        aDNSServer[IP_MAX_DNS_SERVERS];
    U32            OptMask;
  } IPCP;
  struct {
    U8             UserLen;
    U8             abUser[64];
    U8             PassLen;
    U8             abPass[64];
    U16            Prot;
    U32            Data;
    PPP_AUTH_STATE State;
    RESEND_INFO    Resend;
    U32            OptMask;
  } Auth;
  IP_PPP_LINE_DRIVER * pLineDriver;
} IP_PPP_CONTEXT;
```


Member	Description
pfSend	Pointer to a function which sends a packet.
pfTerm	Pointer to a function which terminates the connection.
pfInformUser	Pointer to a callback function which informs the user about a status change of the connection.
pSendContext	Pointer to a user callback function which is triggered when a status change of the PPP connection occurs.
NumBytesPrepend	The size of the PPP header to be prepended when sending packets.
IFaceId	Internal index number of the interface.
Config.NumTries	Defines the number of times the stack tries to initialise a connection via PADI before giving up. Can be set via IP_PPPOE_ConfigRetries(), the default is 5.
Config.Timeout	Sets the timeout between PADI configuration retries in ms, the default is 2000.
LCP.Id	Sequential ID number of the LCP packet.
LCP.aOptCnt	An array of supported LPC options.
LCP.AState	An enum of type PPP_LCP_STATE. Indicates the active status of the LPC connection.
LCP.PState	An enum of type PPP_LCP_STATE. Indicates the passive status (modem side) of the LPC connection.
LCP.Resend	A structure of type RESEND_INFO.
LCP.MRU	Maximum-Receive-Unit.
LCP.ACCM	Async-Control-Character-Map.
LCP.OptMask	Mask to identify the options which should be added to the LCP packet.
CCP.Id	Sequential ID number of the CCP packet.
CCP.aOptCnt	An array of supported CCP options.
CCP.AState	An enum of type PPP_CCP_STATE. Indicates the active status of the CCP connection.
CCP.PState	An enum of type PPP_CCP_STATE. Indicates the passive status (modem side) of the LPC connection.
CCP.Resend	A structure of type RESEND_INFO.
CCP.OptMask	Mask to identify the options which should be added to the CCP packet.
IPCP.Id	Sequential ID number of the IPCP packet.
IPCP.aOptCnt	An array of supported IPCP options.
IPCP.AState	An enum of type PPP_IPCP_STATE. Indicates the active status of the LPC connection.
IPCP.PState	An enum of type PPP_IPCP_STATE. Indicates the passive status (modem side) of the LPC connection.
IPCP.Resend	A structure of type RESEND_INFO.
IPCP.IpAddr	An IP_ADDR to store the IP adress of the PPP interface.
IPCP.aDNSServer	An IP_ADDR to store the IP adress of the PPP interface.
IPCP.OptMask	Mask to identify the options which should be added to the IPCP packet.
Auth.UserLen	Length of the user name, is being set internally.
Auth.abUser	User name for the PPPoE connection.
Auth.PassLen	Length of the user password, is being set internally.
Auth.abPass	User password for the PPPoE connection.
Auth.Prot	Defines the PPP authentication protocol, is set typically to PPP_PROT_PAP.

Table 19.23: Structure IP_PPP_CONTEXT member list

Member	Description
Auth.State	An enum of type PPP_AUTH_STATE.
Auth.Resend	A structure of type RESEND_INFO.
pLineDriver	Pointer to a structure of type IP_PPP_LINE_DRIVER

Table 19.23: Structure IP_PPP_CONTEXT member list

19.9.2 Structure RESEND_INFO

Description

A structure which stores the resend condition for different stages of the PPP connection.

Prototype

```
typedef struct {
  IP_PACKET * pPacket;
  I32         Timeout;
  I32         InitialTimeout;
  U32         RemTries;
#if IP_DEBUG
  const char * sPacketName;
#endif
} RESEND_INFO;
```

Member	Description
pPacket	Pointer to an IP_PACKET structure.
Timeout	Timeout in ms before a resend is triggered.
InitialTimeout	Initial timeout in ms before a resend is triggered. Saved to be able to reset Timeout to it's original state.
RemTries	Counter for the remaining number of retries.
sPacketName	(Only with IP_DEBUG >= 1.) Custom name assigned to the packet.

Table 19.24: Structure RESEND_INFO member list

19.9.3 Structure IP_PPP_LINE_DRIVER

Description

Structure with pointers to application related functions.

Prototype

```
typedef struct {
  void (*pfInit) (struct IP_PPP_CONTEXT * pPPPContext);
  void (*pfSend) (U8 Data);
  void (*pfSendNext) (U8 Data);
  void (*pfTerminate) (U8 IFaceId);
  void (*pfOnPacketCompletion) (void);
} IP_PPP_LINE_DRIVER;
```

Member	Description
pfInit	Pointer to a function which initialises the PPP connection.
pfSend	Pointer to a function which sends the first byte.
pfSendNext	Pointer to a function which sends the next byte. Typically called from an interrupt that confirms that the last byte has been sent.
pfTerminate	Pointer to a function which terminates the connection.
pfOnPacketCompletion	Optional. Called when packet is complete. Normally used for packet oriented PPP interfaces GPRS or USB modems.

Table 19.25: Structure IP_PPP_LINE_DRIVER member list

19.10 PPPoE resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the PPP/PPPoE modules presented in the tables below have been measured on an ARM7 and a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

The resource usage of a typical PPPoE scenario with 1 WAN interface has been measured.

19.10.1 ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP PPP used for PPPoE	approximately 13.9Kbyte

Table 19.26: PPPoE ROM usage ARM7

19.10.2 ROM usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

Addon	ROM
embOS/IP PPP used for PPPoE	approximately 12.2Kbyte

Table 19.27: PPPoE ROM usage Cortex-M3

19.10.3 RAM usage

Addon	RAM
embOS/IP PPP used for PPPoE	approximately 4.4Kbyte

Table 19.28: PPPoE RAM usage

19.11 PPP resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the PPP modules presented in the tables below have been measured on an ARM7 system. Details about the further configuration can be found in the sections of the specific example.

The resource usage of a typical PPP scenario without network interface and one modem connected via RS232 has been measured.

19.11.0.1ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP PPP	approximately 13.4Kbyte

Table 19.29: PPP ROM usage ARM7

19.11.0.2RAM usage

Addon	RAM
embOS/IP PPP	approximately 11.8Kbyte

Table 19.30: PPP RAM usage

Chapter 20

NetBIOS (Add-on)

The embOS/IP implementation of the Network Basic Input/Output System Protocol (NetBIOS) is an optional extension to embOS/IP. It can be used to resolve NetBIOS names in a local area network. All functions that are required to add NetBIOS to your application are described in this chapter.

20.1 embOS/IP NetBIOS

The embOS/IP NetBIOS implementation is an optional extension which can be seamlessly integrated into your TCP/IP application. It combines a maximum of performance with a small memory footprint. The NetBIOS implementation allows an embedded system to resolve NetBIOS names in the local area network.

The NetBIOS module implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 1001]	NetBIOS Concenpts and methods Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1001.txt*
[RFC 1002]	NetBIOS Detailed Specifications Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1002.txt*

The following table shows the contents of the embOS/IP root directory:

Directory	Content
`Application`	Contains the example application to run the NetBIOS implementation with embOS/IP.
`Inc`	Contains the required include files.
`IP`	Contains the NetBIOS sources, IP_Netbios.c.

Supplied directory structure of embOS/IP NetBIOS package

20.2 Feature list

- Low memory footprint.
- Seamless integration with the embOS/IP stack.
- Client based NetBIOS name resolution.

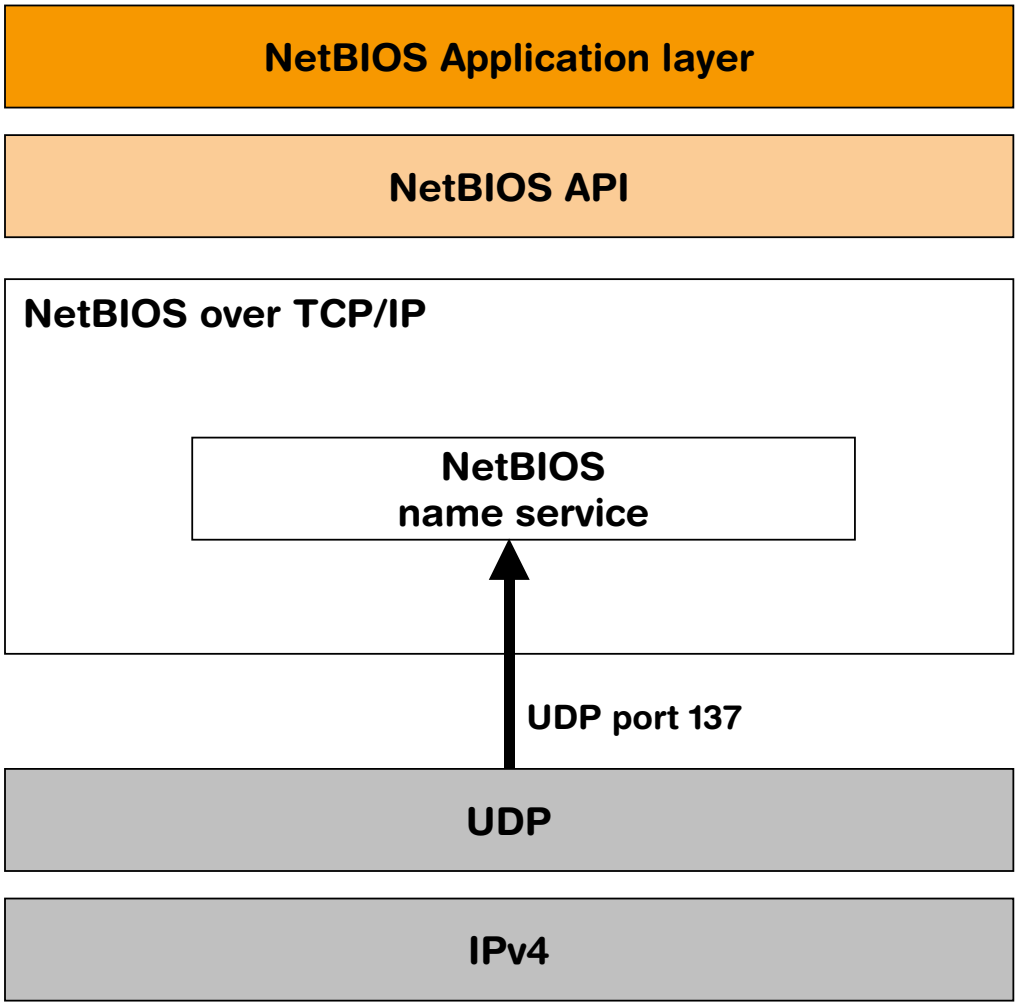
20.3 Requirements

TCP/IP stack

The embOS/IP NetBIOS implementation requires the embOS/IP TCP/IP stack.

20.4 NetBIOS backgrounds

The Network Basic Input/Output System protocol is an API on top of the TCP/IP protocol, it provides a way of communication between separate computers within a local arena network via the session layer.



Using NetBIOS, an embOS/IP application can resolve a NetBIOS name to an IP adress in the local area network.

20.5 API functions

Function	Description
	NetBIOS
IP_NETBIOS_Init()	Initializes the NetBIOS Name Service client.
IP_NETBIOS_Start()	Starts the NetBIOS client.
IP_NETBIOS_Stop()	Stops the NetBIOS client.

Table 20.1: embOS/IP NetBIOS API function overview

20.5.1 IP_NETBIOS_Init()

Description

Initializes the NetBIOS Name Service client.

Prototype

```
int IP_NETBIOS_Init( U32                       IFaceId,
                     const IP_NETBIOS_NAME * paHostnames,
                     U16                       LPort );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.
paHostnames	[IN] Pointer to an array of Structure IP_NETBIOS_NAME. Expects last index to be zero filled.
LPort	[IN] Local port used for listening. Typically 137. If parameter LPort is 0, 137 will be used.

Table 20.2: IP_NETBIOS_Init() parameter list

Return value

< 0: Error, invalid NetBIOS name in paHostnames list.
> 0: Ok, Number of valid NetBIOS names assigned to the target.

20.5.2 IP_NETBIOS_Start()

Description

Starts the NetBIOS client. Creates an UDP socket to receive Netbios Name Service requests.

Prototype

```
int IP_NETBIOS_Start ( U32 IFaceId );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.

Table 20.3: IP_NETBIOS_Start() parameter list

Return value

0: Error, could not create an UDP socket for NetBIOS.
> 0: OK, number of the socket which is used for the NetBIOS Name Service.

20.5.3 IP_NETBIOS_Stop()

Description

Stops the NetBIOS client. Closes the UDP socket.

Prototype

```
void IP_NETBIOS_Stop ( U32 IFaceId );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.

Table 20.4: IP_NETBIOS_Stop() parameter list

20.5.4 Structure IP_NETBIOS_NAME

Description

A structure which stores the information about the NetBIOS name.

Prototype

```
typedef struct IP_NETBIOS_NAME {
  char * sName;
  U8 NumBytes;
} IP_NETBIOS_NAME;
```

Member	Description
sName	[IN] Pointer to a string which stores the NetBIOS name.
NumBytes	[IN] Length of the NetBIOS name without termination.

Table 20.5: Structure IP_NETBIOS_NAME member list

20.6 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the NetBIOS module presented in the tables below have been measured on an ARM7 and a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

20.6.1 ROM usage on an ARM7 system

The following resource usage has been measured on an ARM7 system using IAR Embedded Workbench V6.30.6, Thumb mode, no interwork, size optimization.

Addon	ROM
embOS/IP NetBIOS module	approximately 0.8Kbyte

Table 20.6: NetBIOS ROM usage ARM7

20.6.2 ROM usage on a Cortex-M3 system

The following resource usage has been measured on a Cortex-M3 system using IAR Embedded Workbench V6.30.6, size optimization.

Addon	ROM
embOS/IP NetBIOS module	approximately 0.7Kbyte

Table 20.7: NetBIOS ROM usage Cortex-M3

20.6.3 RAM usage

Addon	RAM
embOS/IP NetBIOS module	approximately 2.4Kbyte

Table 20.8: NetBIOS RAM usage

Chapter 21

Debugging

embOS/IP comes with various debugging options. These includes optional warning and log outputs, as well as other run-time options which perform checks at run time as well as options to drop incoming or outgoing packets to test stability of the implementation on the target system.

21.1 Message output

The debug builds of embOS/IP include a fine grained debug system which helps to analyze the correct implementation of the stack in your application. All modules of the TCP/IP stack can output logging and warning messages via terminal I/O, if the specific message type identifier is added to the log and/or warn filter mask. This approach provides the opportunity to get and interpret only the logging and warning messages which are relevant for the part of the stack that you want to debug.

By default, all warning messages are activated in all embOS/IP sample configuration files. All logging messages are disabled except for the messages from the initialization and the DHCP setup phase.

21.2 Testing stability

embOS/IP allows to define drop-rates for both receiver an transmitter. This feature can be used to simulate packet loss. Packet loss means that one or more packets fail to reach their destination. Packet loss can be caused by a number of factors (for example, signal degradation over the network medium, faulty networking hardware, error in network applications, etc.).

Two variables, `IP_TxDropRate` and `IP_RxDropRate`, are implemented to define the drop-rate while the target is running. There is no need to recompile the stack. The default value of these variables is 0, which means that no packets should be dropped from the stack. Any other value of n (for example, n = 2,3, ...) will drop every n-th packet. This allows testing the reliability of communication and performance drop. A good value to test the stability is typically around 50.

To change the value of `IP_TxDropRate` and/or `IP_RxDropRate` the following steps are required:

1. Download your embOS/IP application into the target.
2. Start your debugger.
3. Open the **Watch** window and add one or both drop-rate variables.
4. Assign the transmit and/or receive drop-rate and start your application.

21.3 API functions

Function	Description
Filter functions	
IP_Log()	This function is called by the stack in debug builds with log & warn output (debug level > 1). In a release build, this function may not be linked in.
IP_Warn()	This function is called by the stack in debug builds with log & warn output (debug level > 1). In a release build, this function may not be linked in.
IP_AddLogFilter()	Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.
IP_AddWarnFilter()	Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.
IP_SetLogFilter()	Sets the mask that defines which logging message should be displayed.
IP_SetWarnFilter()	Sets the mask that defines which warning message should be displayed.
General debug functions/macros	
IP_PANIC()	Called if the stack encounters a critical situation.

Table 21.1: embOS/IP debugging API function overview

21.3.1 IP_AddLogFilter()

Description

Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.

Prototype

```
void IP_AddLogFilter(U32 FilterMask);
```

Parameter

Parameter	Description
FilterMask	Specifies which logging messages should be added to the filter mask. Refer to *Message types* on page 442 for a list of valid values for parameter FilterMask.

Table 21.2: IP_AddLogFilter() parameter list

Additional information

IP_AddLogFilter() can also be used to remove a filter condition which was set before. It adds/removes the specified filter to/from the filter mask via a disjunction.

Example

```
IP_AddLogFilter(IP_MTYPE_DRIVER); // Activate driver logging messages
/*
 * Do something
 */
IP_AddLogFilter(IP_MTYPE_DRIVER); // Deactivate all driver logging messages
```


21.3.2 IP_AddWarnFilter()

Description

Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.

Prototype

```
void IP_AddWarnFilter(U32 FilterMask);
```

Parameter

Parameter	Description
FilterMask	Specifies which warning messages should be added to the filter mask. Refer to *Message types* on page 442 for a list of valid values for parameter FilterMask.

Table 21.3: IP_AddWarnFilter() parameter list

Additional information

IP_AddWarnFilter() can also be used to remove a filter condition which was set before. It adds/removes the specified filter to/from the filter mask via a disjunction.

Example

```
IP_AddWarnFilter(IP_MTYPE_DRIVER); // Activate driver warning messages
/*
 * Do something
 */
IP_AddWarnFilter(IP_MTYPE_DRIVER); // Deactivate all driver warning messages
```


21.3.3 IP_SetLogFilter()

Description

Sets a mask that defines which logging message that should be logged. Logging messages are only available in debug builds of embOS/IP.

Prototype

```
void IP_SetLogFilter( U32 FilterMask );
```

Parameter

Parameter	Description
FilterMask	Specifies which logging messages should be displayed. Refer to *Message types* on page 442 for a list of valid values for parameter FilterMask.

Table 21.4: IP_SetLogFilter() parameter list

Additional information

This function should be called from IP_X_Config(). By default, the filter condition IP_MTYPE_INIT is set. Refer to *IP_X_Configure()* on page 256 for more information.

21.3.4 IP_SetWarnFilter()

Description

Sets a mask that defines which warning messages that should be logged. Warning messages are only available in debug builds of embOS/IP.

Prototype

```
void IP_SetWarnFilter( U32 FilterMask );
```

Parameter

Parameter	Description
FilterMask	Specifies which warning messages should be displayed. Refer to *Message types* on page 442 for a list of valid values for parameter FilterMask.

Table 21.5: IP_SetWarnFilter() parameter list

Additional information

This function should be called from IP_X_Config(). By default, all filter conditions are set. Refer to *IP_X_Configure()* on page 256 for more information.

21.3.5 IP_PANIC()

Description

This macro is called by the stack code when it detects a situation that should not be occurring and the stack can not continue. The intention for the `IP_PANIC()` macro is to invoke whatever debugger may be in use by the programmer. In this way, it acts like an embedded breakpoint.

Prototype

```
IP_PANIC ( const char * sError );
```

Additional information

This macro maps to a function in debug builds only. If `IP_DEBUG > 0`, the macro maps to the stack internal function `void IP_Panic ( const char * sError )`. `IP_Panic()` disables all interrupts to avoid further task switches, outputs `sError` via terminal I/O and loops forever. When using an emulator, you should set a breakpoint at the beginning of this routine or simply stop the program after a failure. The error code is passed to the function as parameter.

In a release build, this macro is defined empty, so that no additional code will be included by the linker.

21.4 Message types

The same message types are used for log and warning messages. Separate filters can be used for both log and warnings. For details, refer to *IP_SetLogFilter()* on page 439 and *IP_SetWarnFilter()* on page 440 as wells as *IP_AddLogFilter()* on page 437 and *IP_AddLogFilter()* on page 437 for more information about using the message types.

Symbolic name	Description
IP_MTYPE_INIT	Activates output of messages from the initialization of the stack that should be logged.
IP_MTYPE_CORE	Activates output of messages from the core of the stack that should be logged.
IP_MTYPE_ALLOC	Activates output of messages from the memory allocating module of the stack that should be logged.
IP_MTYPE_DRIVER	Activates output of messages from the driver that should be logged.
IP_MTYPE_ARP	Activates output of messages from ARP module that should be logged.
IP_MTYPE_IP	Activates output of messages from IP module that should be logged.
IP_MTYPE_TCP_CLOSE	Activates output of messages from TCP module that should be logged when a TCP connection gets closed.
IP_MTYPE_TCP_OPEN	Activates output of messages from TCP module that should be logged when a TCP connection gets opened.
IP_MTYPE_TCP_IN	Activates output of messages from TCP module that should be logged if a TCP packet is received.
IP_MTYPE_TCP_OUT	Activates output of messages from TCP module that should be logged if a TCP packet is sent.
IP_MTYPE_TCP_RTT	Activates output of messages from TCP module regarding TCP roundtrip time.
IP_MTYPE_TCP_RXWIN	Activates output of messages from TCP module regarding peer TCP Rx window size.
IP_MTYPE_TCP	Activates output of messages from TCP that module should be logged.
IP_MTYPE_UDP_IN	Activates output of messages from UDP module that should be logged when a UDP packet is received.
IP_MTYPE_UDP_OUT	Activates output of messages from UDP module that should be logged if a UDP packet is sent.
IP_MTYPE_UDP	Activates output of messages from UDP module that should be logged if a UDP packet is sent or received.
IP_MTYPE_LINK_CHANGE	Activates output of messages regarding to the link change process.
IP_MTYPE_AUTOIP	Activates output of from the AutoIP module that should be logged.
IP_MTYPE_DHCP	Activates output of messages from DHCP client module that should be logged.
IP_MTYPE_DHCP_EXT	Activates output of optional messages from DHCP client module that should be logged.

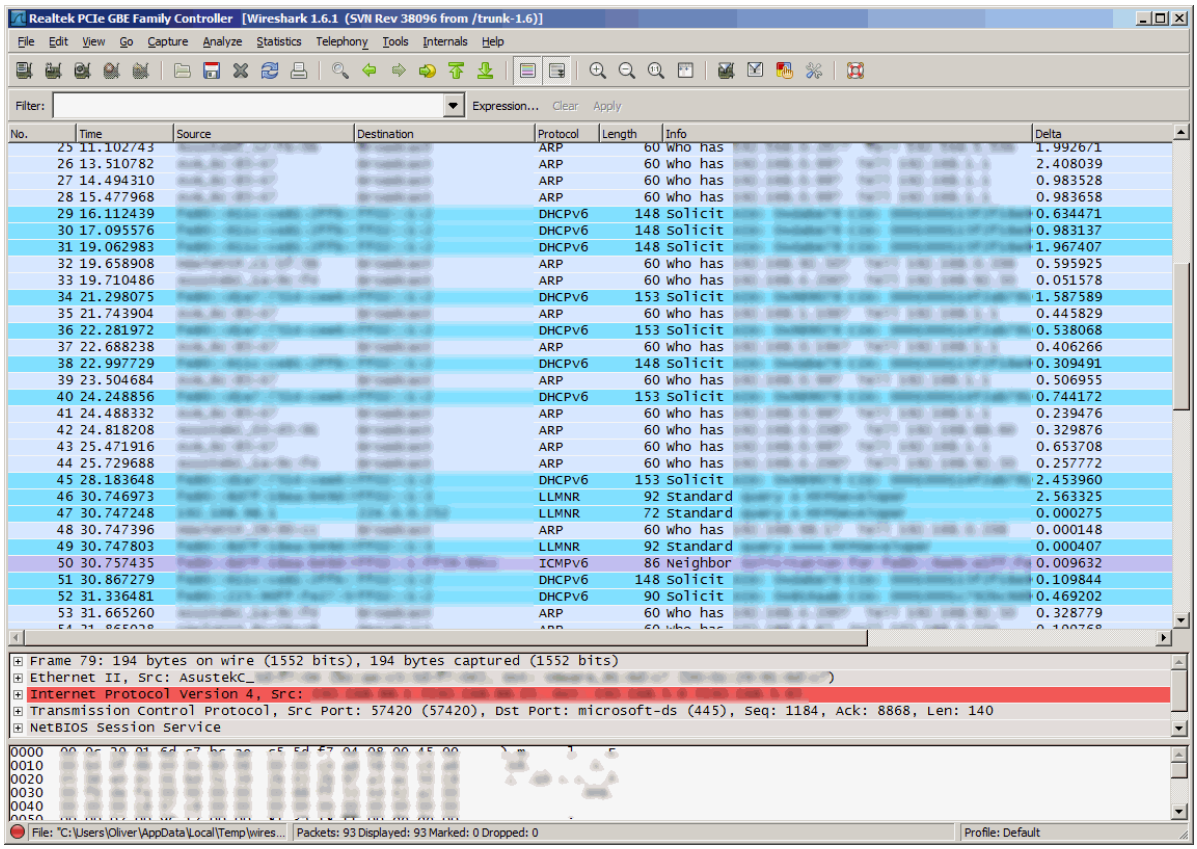
Table 21.6: embOS/IP message types

Symbolic name	Description
IP_MTYPE_APPLICATION	Activates output of messages from user application related modules that should be logged.
IP_MTYPE_ICMP	Activates output of messages from the ICMP module that should be logged.
IP_MTYPE_NET_IN	Activates output of messages from NET_IN module that should be logged.
IP_MTYPE_NET_OUT	Activates output of messages from NET_OUT module that should be logged.
IP_MTYPE_PPP	Activates output of messages from PPP modules that should be logged.
IP_MTYPE_SOCKET_STATE	Activates output of messages from socket module that should be logged when state has been changed.
IP_MTYPE_SOCKET_READ	Activates output of messages from socket module that should be logged if a socket is used to read data.
IP_MTYPE_SOCKET_WRITE	Activates output of messages from socket module that should be logged if a socket is used to write data
IP_MTYPE_SOCKET	Activates all socket related output messages.
IP_MTYPE_DNSC	Activates output of messages from DNS client module that should be logged.
IP_MTYPE_ACD	Activates output of messages from address conflict module that should be logged.

Table 21.6: embOS/IP message types

21.5 Using a network sniffer to analyse communication problems

Using a network sniffer to analyze your local network traffic may give you a deeper understanding of the data that is being sent in your network. For this purpose you can use several network sniffers. Some of them are available for free such as `Wireshark`. An example of a network sniff using `Wireshark` is shown in the screenshot below:



Chapter 22

OS integration

embOS/IP is designed to be used in a multitasking environment. The interface to the operating system is encapsulated in a single file, the IP/OS interface. For embOS, all functions required for this IP/OS interface are implemented in a single file which comes with embOS/IP.

This chapter provides descriptions of the functions required to fully support embOS/IP in multitasking environments.

22.1 General information

The complexity of the IP/OS Interface depends on the task model selected. Refer to *Tasks and interrupt usage* on page 19 for detailed informations about the different task models. All OS interface functions for embOS are implemented in `IP_OS_embOS.c` which is located in the root folder of the IP stack.

22.2 OS layer API functions

Function	Description
General macros	
IP_OS_Delay()	Blocks the calling task for a given time.
IP_OS_DisableInterrupt()	Disables interrupts.
IP_OS_EnableInterrupt()	Enables interrupts.
IP_OS_GetTime32()	Returns the current system time in ticks. Return the current system time in ms. On 32-bit systems, the value will wrap around after approximately 49.7 days. This is taken into account by the stack.
IP_OS_Init()	Creates and initializes all objects required for task synchronization. These are 2 events (for IP_Task and IP_RxTask) and one semaphore for protection of critical code which may not be executed from multiple task at the same time.
IP_OS_Lock()	The stack requires a single lock, typically a resource semaphore or mutex. This function locks this object, guarding sections of the stack code against other tasks. If the entire stack executes from a single task, no functionality is required here.
IP_OS_Unlock()	Unlocks the single lock used locked by a previous call to IP_OS_Lock().
IP_Task synchronization	
IP_OS_SignalNetEvent()	Wakes the IP_Task if it is waiting for a NET-event or timeout in the function IP_OS_WaitNetEvent().
IP_OS_WaitNetEvent()	Called from IP_Task only. Blocks until the timeout expires or a NET-event occurs, meaning IP_OS_SignalNetEvent() is called from an other task or ISR.
IP_RxTask synchronization	
IP_OS_SignalRxEvent()	Wakes the IP_RxTask if it is waiting for a NET-event or timeout in the function IP_OS_WaitRxEvent().
IP_OS_WaitRxEvent()	Optional. Called from IP_RxTask, if it is used to receive data. Blocks until the timeout expires or a NET-event occurs, meaning IP_OS_SignalRxEvent() is called from the ISR.
Application task synchronization	
IP_OS_WaitItem()	Suspend a task which needs to wait for a object. This object is identified by a pointer to it and can be of any type, for example a socket.
IP_OS_WaitItemTimed()	Suspend a task which needs to wait for a object. This object is identified by a pointer to it and can be of any type, for example a socket. The second parameter defines the maximum time in timer ticks until the event have to be signaled.
IP_OS_SignalItem()	Sets an event object to signaled state, or resumes tasks which are waiting at the event object. Function is called from a task, not an ISR.

Table 22.1: Target OS interface function list

22.2.1 Examples

OS interface routine for embOS

All OS interface routines are implemented in IP_OS_embOS.c which is located in the root folder of the IP stack.

Chapter 23

Performance & resource usage

This chapter covers the performance and resource usage of embOS/IP. It contains information about the memory requirements in typical systems which can be used to obtain sufficient estimates for most target systems.

23.1 Memory footprint

embOS/IP is designed to fit many kinds of embedded design requirements. Several features can be excluded from a build to get a minimal system. Note that the values are only valid for the given configurations.

23.1.1 ARM7 system

The following table shows the hardware and the toolchain details of the project:

Detail	Description
CPU	ARM7
Tool chain	IAR Embedded Workbench for ARM V6.30.6
Model	ARM7, Thumb instructions; no interwork;
Compiler options	Highest size optimization;

Table 23.1: ARM7 sample configuration

23.1.1.1 ROM usage

The following table shows the ROM requirement of embOS/IP:

Description	ROM
embOS/IP - complete stack	approximately 19.4Kbytes

The memory requirements of a interface driver is about 1.5 - 2.0Kbytes.

23.1.1.2 RAM usage

The following table shows the RAM requirement of embOS/IP:

Description	RAM
embOS/IP - complete stack w/o buffers	approximately 5.2Kbytes

23.1.2 Cortex-M3 system

The following table shows the hardware and the toolchain details of the project:

Detail	Description
CPU	Cortex-M3
Tool chain	IAR Embedded Workbench for ARM V6.30.6
Model	Cortex-M3
Compiler options	Highest size optimization;

Table 23.2: ARM7 sample configuration

23.1.2.1 ROM usage

The following table shows the ROM requirement of embOS/IP:

Description	ROM
embOS/IP - complete stack	approximately 19Kbytes

The memory requirements of a interface driver is about 1.5 - 2.0Kbytes.

23.1.2.2 RAM usage

The following table shows the RAM requirement of embOS/IP:

Description	RAM
embOS/IP - complete stack w/o buffers	approximately 4.5Kbytes

23.2 Performance

23.2.1 ARM7 system

Detail	Description
CPU	ARM7 with integrated MAC running with 48Mhz
Tool chain	IAR Embedded Workbench for ARM V6.30.6
Model	ARM7, Thumb instructions; no interwork;
Compiler options	Highest speed optimization;

Table 23.3: ARM7 sample configuration

Memory configuration

```
#define ALLOC_SIZE  0xD000
IP_AddBuffers(12, 256);
IP_AddBuffers(18, mtu + 16);
IP_ConfTCPSpace(8 * (mtu-40), 8 * (mtu-40));
```

Driver configuration

```
#define NUM_RX_BUFFERS   (2 * 12 + 1)
```

Measurements

The following table shows the send and receive speed of embOS/IP:

Description	Speed [Mbytes per second]
TCP - socket interface	
Send speed	approximately 9.0
Receive speed	approximately 7.5
TCP - zero-copy interface	
Send speed	approximately 9.0
Receive speed	approximately 11.7

The performance of any network will depend on several considerations, including the length of the cabling, the size of packets, and the amount of traffic.

23.2.2 Cortex-M3 system

Detail	Description
CPU	Cortex-M3 with integrated MAC running with 96Mhz
Tool chain	IAR Embedded Workbench for ARM V6.30.6
Model	Cortex-M3
Compiler options	Highest speed optimization;

Table 23.4: ARM7 sample configuration

Memory configuration

```
#define ALLOC_SIZE  0x10000
IP_AddBuffers(12, 256);
IP_AddBuffers(12, mtu + 16);
IP_ConfTCPSpace(9 * (mtu-40), 9 * (mtu-40));
```

Driver configuration

```
#define NUM_RX_BUFFERS      (36)
#define BUFFER_SIZE         (256)
```

Measurements

The following table shows the send and receive speed of embOS/IP:

Description	Speed [Mbytes per second]
TCP - socket interface	
Send speed	approximately 9.4
Receive speed	approximately 11.7
TCP - zero-copy interface	
Send speed	approximately 9.4
Receive speed	approximately 11.8

The performance of any network will depend on several considerations, including the length of the cabling, the size of packets, and the amount of traffic.

Chapter 24

Appendix A - File system abstraction layer

24.1 File system abstraction layer

This section provides a description of the file system abstraction layer used by embOS/IP applications which require access to a data storage medium. The file system abstraction layer is supplied with the embOS/IP web server and the embOS/IP FTP server.

Three file system abstraction layer implementations are available:

File name	Description
IP_FS_FS.c	Mapping of the embOS/IP file system abstraction layer functions to the emFile functions.
IP_FS_RO.c	Implementation of a read-only file system. Typically used in a web server application.
IP_FS_WIN32.c	Mapping of the embOS/IP file system abstraction layer functions to the Windows file I/O functions.

Supplied implementations of the file system abstraction layer

24.2 File system abstraction layer function table

embOS/IP uses a function table to call the appropriate file system function.

Data structure

```
typedef struct {
  //
  // Read only file operations. These have to be present on ANY file system,
  // even the simplest one.
  //
  void * (*pfOpenFile)   ( const char * sFilename,
                           const char * sOpenFlags );
  int    (*pfCloseFile)  (       void * hFile );
  int    (*pfReadAt)     (       void * hFile,
                                 void * pBuffer,
                                 U32    Pos,
                                 U32    NumBytes );
  long   (*pfGetLen)     (       void * hFile );
  //
  // Directory query operations.
  //
  void   (*pfForEachDirEntry)       (       void * pContext,
                                      const char * sDir,
                                            void (*pf) (void * pContext,
                                                        void * pFileEntry));
  void   (*pfGetDirEntryFileName)   (       void * pFileEntry,
                                            char * sFileName,
                                            U32    SizeOfBuffer );
  U32    (*pfGetDirEntryFileSize)   (       void * pFileEntry,
                                            U32  * pFileSizeHigh );
  int    (*pfGetDirEntryFileTime)   (       void * pFileEntry );
  U32    (*pfGetDirEntryAttributes) (       void * pFileEntry );
  //
  // Write file operations.
  //
  void * (*pfCreate)     ( const char * sFileName );
  void * (*pfDeleteFile) ( const char * sFilename );
  int    (*pfRenameFile) ( const char * sOldFilename,
                           const char * sNewFilename );
  int    (*pfWriteAt)    ( void       * hFile,
                           void       * pBuffer,
                           U32          Pos,
                           U32          NumBytes );
  //
  // Additional directory operations
  //
  int    (*pfMKDir)      (const char * sDirName);
  int    (*pfRMDir)      (const char * sDirName);
} IP_FS_API;
```


Elements of IP_FS_API

Function	Description
Read only file system functions (required)	
pfOpenFile	Pointer to a function that creates/opens a file and returns the handle of these file.
pfCloseFile	Pointer to a function that closes a file.
pfReadAt	Pointer to a function that reads a file.
pfGetLen	Pointer to a function that returns the length of a file.
Directory query operations	
pfForEachDirEntry	Pointer to a function which is called for each directory entry.
pfGetDirEntryFileName	Pointer to a function that returns the name of a file entry.
pfGetDirEntryFileSize	Pointer to a function that returns the size of a file.
pfGetDirEntryFileTime	Pointer to a function that returns the time-stamp of a file.
pfGetDirEntryAttributes	Pointer to a function that returns the attributes of a directory entry.
Write file operations	
pfCreate	Pointer to a function that creates a file.
pfDeleteFile	Pointer to a function that deletes a file.
pfRenameFile	Pointer to a function that renames a file.
pfWriteAt	Pointer to a function that writes a file.
Simple write type file system functions (optional)	
pfCreate	Pointer to a function that creates a file.
pfDeleteFile	Pointer to a function that deletes a file.

Table 24.1: embOS/IP file system API function overview

24.2.1 emFile interface

The embOS/IP web server and FTP server are shipped with an interface to emFile, SEGGER's file system for embedded applications. It is a good example how to use a real file system with the embOS/IP web server / FTP server.

```
/* Excerpt from IP_FS_FS.c */

const IP_FS_API IP_FS_FS = {
  _FS_Open,
  _Close,
  _ReadAt,
  _GetLen,
  _ForEachDirEntry,
  _GetDirEntryFileName,
  _GetDirEntryFileSize,
  _GetDirEntryFileTime,
  _GetDirEntryAttributes,
  _Create,
  _DeleteFile,
  _RenameFile,
  _WriteAt
};
```

The emFile interface is used in all SEGGER Eval Packages.

24.2.2 Read-only file system

The embOS/IP web server and FTP server are shipped with a very basic implementation of a read-only file system. It is a good solution if you use embOS/IP without a real file system like emFile.

```
/* Excerpt from FS_RO.c */

const IP_WEBS_FS_API IP_FS_ReadOnly = {
  _FS_RO_FS_Open,
  _FS_RO_Close,
  _FS_RO_ReadAt,
  _FS_RO_GetLen,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL
};
```

The read-only file system can be used in the example applications. It is sufficient, if the server should only deliver predefined files which are hardcoded in the sources of your application. It is used by default with the embOS/IP Web server example application.

24.2.3 Using the read-only file system

The read-only file system relies on an array of directory entries. A directory entry consists of a file name, a pointer to the data and an entry for the file size in bytes. This array of directory entries will be searched if a client requests a page.

```
/* Excerpt from FS_RO.c */
typedef struct {
  const char * sFilename;
  const unsigned char * pData;
  unsigned int FileSize;
} DIR_ENTRY;

#include "webdata\generated\embos.h"      /* HTML page */
#include "webdata\generated\index.h"      /* HTML page */
#include "webdata\generated\segger.h"     /* segger.gif */
#include "webdata\generated\stats.h"      /* HTML page */

DIR_ENTRY _aFile[] =  {
  /* file name     file array    current size  */
  /* ---------     ----------    ------------  */
  { "/embos.htm",   embos_file,   EMBOS_SIZE   },
  { "/index.htm",   index_file,   INDEX_SIZE   },
  { "/segger.gif",  segger_file,  SEGGER_SIZE  },
  { "/stats.htm",   stats_file,   STATS_SIZE   },
  { 0 }
};
```

The example source files can easily be replaced. To build new contents for the read-only file system the following steps are required:

1. Copy the file which should be included in the read-only file system into the folder: `IP\IP_FS\FS_RO\webdata\html\`
2. Use an text editor (for example, Notepad) to edit the batch file m.bat. The batch file is located under: `IP\IP_FS\FS_RO\webdata\`. Add the file which should be built. For example: If your file is called `example.htm`, you have to add the following line to `m.bat`:
   `call cc example htm`
3. `m.bat` calls `cc.bat`. `cc.bat` uses `bin2C.exe` an utility which converts any file to a standard C array. The new files are created in the folder:
   `IP\IP_FS\FS_RO\webdata\generated\`

4. Add the new source code file (for example, `example.c`) into your project. To add the new file to your read-only file system, you have to add the new file to the `DIR_ENTRY` array `_aFile[]` and include the generated header file (for example, `example.h`) in `FS_RO.c`.
   The expanded definition of `_aFile[]` should look like:

```
#include "webdata\generated\embos.h"      /* HTML page */
#include "webdata\generated\index.h"      /* HTML page */
#include "webdata\generated\segger.h"     /* segger.gif */
#include "webdata\generated\stats.h"      /* HTML page */
#include "webdata\generated\example.h"    /* NEW HTML page */

DIR_ENTRY _aFile[] =  {
  /* file name     file array    current size  */
  /* ---------     ----------    ------------  */
  { "/embos.htm",   embos_file,   EMBOS_SIZE   },
  { "/index.htm",   index_file,   INDEX_SIZE   },
  { "/segger.gif",  segger_file,  SEGGER_SIZE  },
  { "/stats.htm",   stats_file,   STATS_SIZE   },
  { "/example.htm", example_file, EXAMPLE_SIZE },
  { 0 }
};
```

5. Recompile your application.

24.2.4 Windows file system interface

The embOS/IP web server and FTP server is shipped with an implementation.

```
const IP_FS_API IP_FS_Win32 = {
  //
  // Read only file operations.
  //
  _IP_FS_WIN32_Open,
  _IP_FS_WIN32_Close,
  _IP_FS_WIN32_ReadAt,
  _IP_FS_WIN32_GetLen,
  //
  // Simple directory operations.
  //
  _IP_FS_WIN32_ForEachDirEntry,
  _IP_FS_WIN32_GetDirEntryFileName,
  _IP_FS_WIN32_GetDirEntryFileSize,
  _IP_FS_WIN32_GetDirEntryFileTime,
  _IP_FS_WIN32_GetDirEntryAttributes,
  //
  // Simple write type file operations.
  //
  _IP_FS_WIN32_Create,
  _IP_FS_WIN32_DeleteFile,
  _IP_FS_WIN32_RenameFile,
  _IP_FS_WIN32_WriteAt,
  //
  // Additional directory operations
  //
  _IP_FS_WIN32_MakeDir,
  _IP_FS_WIN32_RemoveDir
};
```

The Windows file system interface is supplied with the FTP and the Web server add-on packages. It is used by default with the embOS/IP FTP server application.

Chapter 25

Glossary

ARP	Address Resolution Protocol.
CPU	Central Processing Unit. The “brain” of a microcontroller; the part of a processor that carries out instructions.
DHCP	Dynamic Host Configuration Protocol.
DNS	Domain Name System.
EOT	End Of Transmission.
FIFO	First-In, First-Out.
FTP	File Transfer Protocol.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol.
ICMP	Internet Control Message Protocol.
IP	Internet Protocol.
ISR	Interrupt Service Routine. The routine is called automatically by the processor when an interrupt is acknowledged. ISRs must preserve the entire context of a task (all registers).
LAN	Local Area Network.
MAC	Media Access Control.
NIC	Network Interface Card.
PPP	Point-to-Point Protocol.
RFC	Request For Comments.
RIP	Routing Information Protocol.
RTOS	Real-time Operating System.
Scheduler	The program section of an RTOS that selects the active task, based on which tasks are ready to run, their relative priorities, and the scheduling system being used.
SLIP	Serial Line Internet Protocol.
SMTP	Simple Mail Transfer Protocol.

Stack	An area of memory with LIFO storage of parameters, automatic variables, return addresses, and other information that needs to be maintained across function calls. In multitasking systems, each task normally has its own stack.
Superloop	A program that runs in an infinite loop and uses no real-time kernel. ISRs are used for real-time parts of the software.
Task	A program running on a processor. A multitasking system allows multiple tasks to execute independently from one another.
TCP	Transmission Control Protocol.
TFTP	Trivial File Transfer Protocol.
Tick	The OS timer interrupt. Usually equals 1 ms.
UDP	User Datagram Protocol.

Index

L

N

O

S

T

U

Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

Segger Microcontroller:

7.50.04