

embOS/IP

CPU independant TCP/
IP stack for embedded
applications

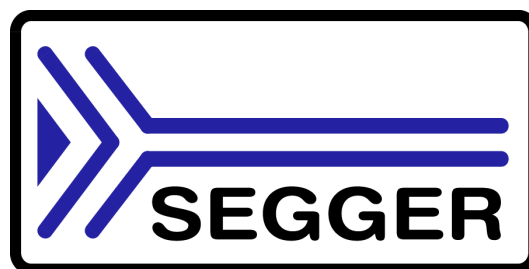
User & Reference Guide

Document: UM07001

Software version: 3.08

Revision: 0

Date: June 30, 2016



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

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2007 - 2016 SEGGER Microcontroller GmbH & Co. KG, Hilden / Germany

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: June 30, 2016

Software	Revision	Date	By	Description
3.08	0	160630	OO	Chapter "Core functions" updated. * IP_AddEtherTypeHook() added. * IP_AddOnPacketFreeHook() added. * IP_AllocEtherPacket() added. * IP_AllocEx() added. * IP_BSP_SetAPI() added. * IP_FreePacket() added. * IP_GetIPAddr() updated. * IP_PHY_DisableCheck() updated. * IP_PHY_DisableCheckEx() updated. * IP_SendEtherPacket() added. * IP_SYSVIEW_Init() added. Chapter "Socket interface" updated. * select() updated. Chapter "Internet Protocol version 6 (IPv6) (Add-on)" updated. * IP_IPV6_Add() updated. Chapter "Web server (Add-on)" updated. * IP_WEBS_AddPreContentOutputHook() added. * IP_WEBS_ConfigUploadRootPath() added. * IP_WEBS_Init() description in API table updated. * IP_WEBS_SendLocationHeader() added. * IP_WEBS_SetHeaderCacheControl() added. Chapter "Profiling with SystemView" added.
3.06	0	160511	OO	Chapter "Core functions" updated. * IP_NI_GetTxQueueLen() added. * IP_STATS module added. Chapter "SNMP agent (Add-on)" added.
3.04a	0	160419	OO	Chapter "Core functions" updated. * IP_FindIFaceByIP() added. * IP_NI_GetAdminState() added. * IP_NI_GetIFaceType() added. * IP_NI_GetState() added. * IP_NI_SetAdminState() added.
3.04	0	160316	OO	Chapter "Configuring embOS/IP" updated. * IP_SUPPORT_TRACE added to compile time switches. Chapter "Core functions" updated. * IP_PHY_DisableCheck() updated. * IP_PHY_DisableCheckEx() updated. * IP_TCP_SetConnKeepaliveOpt() updated. Chapter "Web server (Add-on)" updated. * IP_WEBS_AddRequestNotifyHook() added.
3.02b	0	151223	OO	Chapter "Core functions" updated. * IP_GetFreePacketCnt() added. * IP_GetIFaceHeaderSize() added. * IP_PHY_ConfigAltAddr() added. * IP_PHY_ConfigUseStaticFilters() added. * IP_PHY_ReInit() added. Chapter "PHY drivers" updated. * IP_PHY_MICREL_SWITCH_ConfigLearnDisable() added. * IP_PHY_MICREL_SWITCH_ConfigRxEnable() added. * IP_PHY_MICREL_SWITCH_ConfigTxEnable() added.

Software	Revision	Date	By	Description
3.02	0	151125	OO	Chapter "Introduction to embOS/IP" updated. * Minor changes. Chapter "Core functions" updated. * IP_ARP_CleanCache() added. * IP_ARP_CleanCacheByInterface() added. * IP_ConfigMaxIFaces() added. * IP_ConfigNumLinkUpProbes() added. * IP_PHY_AddDriver() added. * IP_PHY_ConfigAddr() added. * IP_PHY_SupportedModes() added. * IP_PHY_DisableCheckEx() added. Chapter "Web server (Add-on)" updated. * IP_WEBS_UseRawEncoding() added. * IP_WEBS_GetConnectInfo() added. Chapter "PHY drivers" added. Chapter "Tail Tagging (Add-on)" added.
3.00a	0	151007	OO	Chapter "Introduction to embOS/IP" updated. * Updated guidelines for task priorities. Chapter "Running embOS/IP on target hardware" updated. * Added information regarding IP\ASM folder. Chapter "DHCP client" updated. * Minor changes. Chapter "Core functions" updated. * IP_ConfigNumLinkUpProbes() added. Chapter "Socket interface" updated. * Added sample to accept(). * Added sample to getpeername(). * IP_SOCKET_GetAddrFam() added. * IP_SOCKET_GetLocalPort() added.
3.00	0	150813	OO	Chapter "Core functions" updated. * IP_GetMaxAvailPacketSize() added. * IP_GetMTU() added. * IP_IGMP_AddEx() added. Chapter "Internet Protocol version 6 (IPv6) (Add-on)" added. Chapter "TCP zero-copy interface" updated. * IP_TCP_AllocEx() added. Chapter "Web server (Add-on)" updated. * IP_WEBS_AddUpload() added. * IP_WEBS_ConfigBufSizes() added. * IP_WEBS_ConfigRootPath() added. * IP_WEBS_Flush() added. * IP_WEBS_Init() added. * IP_WEBS_ProcessEx() added. * IP_WEBS_ProcessLastEx() added. * IP_WEBS_SendHeaderEx() added.
2.20h	0	150616	OO	Chapter "Core functions" updated. * IP_SetPacketToS() added. Chapter "Socket interface" updated. * Description and prototype of getsockname() updated. Chapter "DHCP client" updated. * IP_DHCP_ConfigAlwaysStartInit() added.
2.20g	0	141223	OO	Chapter "Core functions" updated. * IP_AddVirtEthernetInterface() added. Chapter "TFTP client/server" updated. * Corrected API table.
2.20f	0	141124	OO	Chapter "UDP zero-copy interface" updated. * Information regarding endianness of parameters updated. Chapter "SMTP client (Add-on)" updated. * Corrected supported authentication from AUTH to LOGIN.
2.20e	0	141031	OO	Chapter "Core functions" updated. * Information for IP_GetAddrMask() corrected. * Information for IP_ResolveHost() updated. * Information for IP_TCP_SetConnKeepaliveOpt() updated. Chapter "Socket interface" updated. * Information for connect() updated.

Software	Revision	Date	By	Description
2.20b	0	141002	OO	Chapter "Core functions" updated. * IP_AddMemory() added. * IP_CACHE_SetConfig() added. * IP_PHY_AddResetHook() added. * IP_PHY_DisableCheck() added. * IP_PHY_SetWdTimeout() added. * IP_UDP_AddEchoServer() added. Chapter "DHCP client" updated. * IP_DHCPC_SetClientId() added. Chapter "UDP zero-copy interface" updated. * Additional information for IP_UDP_Send() updated.
2.20	0	140430	OO	Chapter "Core functions" updated. * IP_ConfigOffCached2Uncached() added. * IP_AddLoopbackInterface() added. * IP_AddStateChangeHook() added. * IP_Alloc() added. * IP_ARP_ConfigMaxPending() added. * IP_Connect() added. * IP_DisableIPRxChecksum() added. * IP_Disconnect() added. * IP_DNS_SetServerEx() added. * IP_EnableIPRxChecksum() added. * IP_Err2Str() added. * IP_Free() added. * IP_GetPrimaryIFace() added. * IP_IsExpired() added. * IP_ResolveHost() added. * IP_SetIFaceConnectHook() added. * IP_SetIFaceDisconnectHook() added. * IP_SetPrimaryIFace() added. * IP_SOCKET_ConfigSelectMultiplier() added. * IP_ICMP_DisableRxChecksum() added. * IP_ICMP_EnableRxChecksum() added. * IP_TCP_DisableRxChecksum() added. * IP_TCP_EnableRxChecksum() added. * IP_UDP_DisableRxChecksum() added. * IP_UDP_EnableRxChecksum() added. * IP_ConfTCPSpace() renamed to IP_ConfigTCPSpace() Chapter "Socket interface" updated. * gethostbaname() parameter changed to "const char *" for standard BSD socket compatibility. Chapter "UDP zero-copy interface" updated. * IP_UDP_GetDestAddr() added. * IP_UDP_GetIFIndex() added. * IP_UDP_GetSrcAddr() added. Chapter "RAW zero-copy interface" updated. * IP_RAW_GetDataSize() added. * IP_RAW_GetDestAddr() added. * IP_RAW_GetIFIndex() added. Chapter "DHCP client" updated. * IP_DHCPC_ConfigOnActivate() added. * IP_DHCPC_ConfigOnFail() added. * IP_DHCPC_ConfigOnLinkDown() added. * IP_DHCPC_Renew() added. Chapter "PPP / PPPoE (Add-on)" updated. * IP_PPP_OnTxChar() return value changed. Chapter "Appendix A - File system application layer" updated. * pfIsFolder added to IP_FS_API structure. * pfMove added to IP_FS_API structure. Chapter "DHCP server (Add-on)" added. Chapter "Performance & resource usage" updated. * Values for ROM & RAM usage updated. Minor changes.
2.12g	0	131216	OO	Chapter "Core functions" updated. * IP_ConfigOffCached2Uncached() added.
2.12f	0	130909	OO	Chapter "Core functions" updated. * IP_AddAfterInitHook() added. Chapter "UDP zero-copy interface" updated. * IP_UDP_GetDataSize() added.

Software	Revision	Date	By	Description
2.12c	0	130515	OO	Chapter "Introduction to embOS/IP" updated. * Added information regarding task priorities. Chapter "Core functions" updated. * Added extended information to IP_DeInit() description. Chapter "Web server (Add-on)" updated. * IP_WEBS_GetURI() added. * IP_WEBS_Reset() added.
2.12b	0	130419	OO	Chapter "FTP client (Add-on)" updated. * DELE command added for IP_FTPC_ExecCmd() .
2.12	0	130312	OO	Minor updates and corrections. Chapter "Core functions" updated. * IP_PHY_DisableCheck() added. * IP_RAW_Add() added. * IP_DNS_GetServer() added. * IP_DNS_GetServerEx() added. Chapter "Socket interface" updated. * Information regarding usage of RAW sockets added. Chapter "Web server (Add-on)" updated. * IP_WEBS_AddVFileHook() updated. * IP_WEBS_Redirect() added. * IP_WEBS_StoreUserContext() added. * IP_WEBS_RetrieveUserContext() added. * IP_WEBS_GetDecodedStrLen() added. * IP_WEBS_METHOD * API added. Chapter "RAW zero-copy interface" added. Chapter "SNTP client" added.
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 multiple 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.

Software	Revision	Date	By	Description
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.
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.

Software	Revision	Date	By	Description
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.
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_DHCP_Activate() updated. Chapter "Debugging": * Section "Testing stability" added. Chapter "Socket interface": * Section "Error codes" added.

Software	Revision	Date	By	Description
1.22	0	071114	SK	<p>Chapter "Introduction":</p> <ul style="list-style-type: none"> * "Request for comments" enhanced. <p>Chapter "API functions":</p> <ul style="list-style-type: none"> * IP_AddLogFilter() added. * IP_AddWarnFilter() added. * IP_GetCurrentLinkSpeed() added. * IP_TCP_Set2MSLDelay() added. * select() added. <p>Various function descriptions enhanced.</p> <p>Chapter "API functions" renamed to "core functions".</p> <p>Socket functions removed from chapter "API functions"</p> <p>Chapter "Socket interface" added.</p> <p>Chapter "DHCP" added.</p> <p>Chapter "UDP zero copy" added.</p> <p>Chapter "TCP zero copy" added.</p> <p>Chapter "Glossary" added.</p> <p>Chapter "Index" updated.</p>
1.00	2	071017	SK	<p>Chapter "Introduction":</p> <ul style="list-style-type: none"> * Section "Features" enhanced. * Section "Basic concepts" added. * Section "Task and interrupt usage" added. * Section "Further readings" added. <p>Chapter "Running embOS/IP" enhanced.</p> <p>Chapter "API functions":</p> <ul style="list-style-type: none"> * 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. <p>Chapter "Device driver".</p> <ul style="list-style-type: none"> * General information updated. * Section "Writing your own driver" added. <p>Chapter "Debugging" added.</p> <p>Chapter "Performance and resource usage" added.</p> <p>Chapter "OS integration" updated.</p>
1.00	1	071002	SK	<p>Product name changed to "embOS/IP":</p> <p>Chapter "API functions":</p> <ul style="list-style-type: none"> * 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: Typographic conventions



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 Compact-Flash 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

1	Introduction to embOS/IP.....	21
1.1	What is embOS/IP	22
1.2	Features.....	22
1.3	Basic concepts	23
1.3.1	embOS/IP structure	23
1.3.2	Encapsulation.....	24
1.4	Tasks and interrupt usage.....	25
1.5	Background information	28
1.5.1	Components of an Ethernet system.....	28
1.6	Further reading	31
1.6.1	Request for Comments (RFC)	31
1.6.2	Related books	32
1.7	Development environment (compiler).....	33
2	Running embOS/IP on target hardware.....	35
2.1	Step 1: Open an embOS start project.....	37
2.2	Step 2: Adding embOS/IP to the start project.....	38
2.3	Step 3: Build the project and test it	40
3	Example applications	41
3.1	Overview	42
3.1.1	embOS/IP DNS client (OS_IP_DNSClient.c)	43
3.1.2	embOS/IP non-blocking connect (OS_IP_NonBlockingConnect.c).....	43
3.1.3	embOS/IP ping (OS_IP_Ping.c).....	43
3.1.4	embOS/IP shell (OS_IP_Shell.c)	43
3.1.5	embOS/IP simple server (OS_IP_SimpleServer.c)	44
3.1.6	embOS/IP speed client (OS_IP_SpeedClient_TCP.c)	44
3.1.7	embOS/IP start (OS_IP_Start.c)	45
3.1.8	embOS/IP UDP discover (OS_IP_UDPDiscover.c / OS_IP_UDPDiscoverZeroCopy.c)	45
4	Core functions.....	47
4.1	API functions	48
4.2	Configuration functions.....	53
4.3	Management functions	117
4.4	Network interface configuration and handling functions.....	123
4.5	PHY configuration functions.....	127
4.6	Statistics functions.....	141
4.7	Other IP stack functions	156
4.8	Stack internal functions, variables and data-structures	201
5	Socket interface	207
5.1	API functions	208
5.2	Socket data structures	238
5.3	Error codes.....	242
6	TCP zero-copy interface	243
6.1	TCP zero-copy	244
6.1.1	Allocating, freeing and sending packet buffers	244

6.1.2	Callback function.....	244
6.2	Sending data with the TCP zero-copy API	245
6.2.1	Allocating a packet buffer	245
6.2.2	Filling the allocated buffer with data	245
6.2.3	Sending the packet.....	245
6.3	Receiving data with the TCP zero-copy API	246
6.3.1	Writing a callback function.....	246
6.3.2	Registering the callback function	246
6.4	API functions	247
7	UDP zero-copy interface.....	253
7.1	UDP zero-copy	254
7.1.1	Allocating, freeing and sending packet buffers.....	254
7.1.2	Callback function.....	254
7.2	Sending data with the UDP zero-copy API	255
7.2.1	Allocating a packet buffer	255
7.2.2	Filling the allocated buffer with data	255
7.2.3	Sending the packet.....	255
7.3	Receiving data with the UDP zero-copy API	256
7.3.1	Writing a callback function.....	256
7.3.2	Registering the callback function	256
7.4	API functions	257
8	RAW zero-copy interface	273
8.1	RAW zero-copy	274
8.1.1	Allocating, freeing and sending packet buffers.....	274
8.1.2	Callback function.....	274
8.2	Sending data with the RAW zero-copy API	275
8.2.1	Allocating a packet buffer	275
8.2.2	Filling the allocated buffer with data	275
8.2.3	Sending the packet.....	275
8.3	Receiving data with the RAW zero-copy API	277
8.3.1	Writing a callback function.....	277
8.3.2	Registering the callback function	277
8.4	API functions	278
9	DHCP client	291
9.1	DHCP backgrounds	292
9.2	API functions	293
10	DHCP server (Add-on)	305
10.1	DHCP backgrounds	306
10.2	API functions	307
10.3	DHCP server resource usage	315
10.3.1	ROM usage on an ARM7 system	315
10.3.2	ROM usage on a Cortex-M3 system	315
10.3.3	RAM usage	315
11	AutoIP	317
11.1	embOS/IP AutoIP backgrounds	318
11.2	API functions	319
11.3	AutoIP resource usage	324
11.3.1	ROM usage on an ARM7 system	324
11.3.2	ROM usage on a Cortex-M3 system	324
11.3.3	RAM usage	324
12	Address Collision Detection	325
12.1	embOS/IP ACD backgrounds.....	326
12.2	API functions	327

12.3	ACD data structures.....	330
12.4	ACD resource usage.....	331
12.4.1	ROM usage on an ARM7 system.....	331
12.4.2	ROM usage on a Cortex-M3 system.....	331
12.4.3	RAM usage	331
13	UPnP (Add-on).....	333
13.1	embOS/IP UPnP	334
13.2	Feature list	335
13.3	Requirements.....	336
13.4	UPnP backgrounds	337
13.4.1	Using UPnP to advertise your service in the network.....	337
13.5	API functions	345
13.6	UPnP resource usage.....	347
13.6.1	ROM usage on an ARM7 system.....	347
13.6.2	ROM usage on a Cortex-M3 system.....	347
13.6.3	RAM usage	347
14	VLAN.....	349
14.1	embOS/IP VLAN	350
14.2	Feature list	351
14.3	VLAN backgrounds.....	352
14.4	API functions	353
14.5	VLAN resource usage	355
14.5.1	ROM usage on an ARM7 system.....	355
14.5.2	ROM usage on a Cortex-M3 system.....	355
14.5.3	RAM usage	355
15	Tail Tagging (Add-on)	357
15.1	embOS/IP Tail Tagging support	358
15.2	Feature list	359
15.3	Use cases for Tail Tagging	360
15.4	Requirements.....	361
15.4.1	Software requirements	361
15.4.2	Hardware requirements	361
15.5	Tail Tagging backgrounds	362
15.6	Optimal MTU and buffer sizes	363
15.7	API functions	364
15.8	Resource usage	369
15.8.1	ROM usage on a Cortex-M4 system.....	369
15.8.2	RAM usage	369
16	Network interface drivers	371
16.1	General information	372
16.1.1	MAC address filtering	372
16.1.2	Checksum computation in hardware.....	372
16.1.3	Ethernet CRC computation	372
16.2	Available network interface drivers.....	373
16.2.1	ATMEL AT91CAP9	374
16.2.2	ATMEL AT91RM9200	379
16.2.3	ATMEL AT91SAM7X.....	383
16.2.4	ATMEL AT91SAM9260	387
16.2.5	DAVICOM DM9000/DM9000A	390
16.2.6	FREESCALE ColdFire MCF5329.....	393
16.2.7	NXP LPC17xx	396
16.2.8	NXP LPC23xx / 24xx	398
16.2.9	ST STR912	400
16.3	Writing your own driver.....	402
16.3.1	Device driver functions	404

17	PHY drivers.....	409
17.1	General information.....	410
17.1.1	When is a specific PHY driver required?	410
17.2	Available PHY drivers	411
18	Configuring embOS/IP	421
18.1	Runtime configuration.....	422
18.1.1	IP_X_Configure()	422
18.1.2	Driver handling	423
18.1.3	Memory and buffer assignment	424
18.2	Compile-time configuration.....	426
18.2.1	Compile-time configuration switches	426
18.2.2	Debug level	427
19	Internet Protocol version 6 (IPv6)	
(Add-on)	429
19.1	embOS/IP IPv6	430
19.2	Feature list.....	431
19.3	IPv6 backgrounds	432
19.3.1	IPv6 address types.....	434
19.3.2	Further reading	436
19.4	Include IPv6 to your embOS/IP start project	437
19.4.1	Open an embOS/IP project and compile it	437
19.4.2	Add the embOS/IP IPv6 add-on to the start project	437
19.4.3	Build the project and test it	439
19.5	Configuration.....	440
19.5.1	Compile time configuration	440
19.5.2	Compile time configuration switches	440
19.5.3	Runtime configuration.....	440
19.6	API functions	441
19.7	Socket API extensions.....	449
19.8	Porting an IPv4 application to IPv6	451
19.8.1	Porting an IPv4 server application to IPv6.....	451
19.9	IPv6 resource usage	460
19.9.1	ROM usage.....	460
19.9.2	RAM usage	460
20	Web server (Add-on).....	461
20.1	embOS/IP Web server	462
20.2	Feature list.....	463
20.3	Requirements	464
20.4	HTTP backgrounds.....	465
20.4.1	HTTP communication basics.....	465
20.4.2	HTTP status codes	466
20.5	Using the Web server sample.....	467
20.5.1	Using the Windows sample	468
20.5.2	Running the Web server example on target hardware.....	468
20.5.3	Changing the file system type	469
20.6	Dynamic content	470
20.6.1	Common Gateway Interface (CGI)	470
20.6.2	Virtual files.....	472
20.6.3	AJAX	475
20.6.4	Server-Sent Events (SSE)	477
20.7	Authentication	480
20.7.1	Authentication example.....	481
20.7.2	Configuration of the authentication	482
20.8	Form handling	483
20.8.1	Simple form processing sample	484
20.9	File upload	487

20.9.1	Simple form upload sample.....	487
20.10	Configuration	489
20.10.1	Compile time configuration	489
20.10.2	Compile time configuration switches.....	489
20.10.3	Runtime configuration	491
20.11	API functions	492
20.12	Web server data structures.....	547
20.13	Resource usage	562
20.13.1	ROM usage on an ARM7 system.....	562
20.13.2	ROM usage on a Cortex-M3 system.....	562
20.13.3	RAM usage:	562
21	SMTP client (Add-on).....	563
21.1	embOS/IP SMTP client.....	564
21.2	Feature list	565
21.3	Requirements.....	566
21.4	SMTP backgrounds.....	567
21.5	Configuration	569
21.5.1	Compile time configuration switches.....	569
21.6	API functions	570
21.7	SMTP client data structures.....	572
21.8	Resource usage	580
21.8.1	Resource usage on an ARM7 system	580
21.8.2	Resource usage on a Cortex-M3 system.....	580
22	FTP server (Add-on)	581
22.1	embOS/IP FTP server	582
22.2	Feature list	583
22.3	Requirements.....	584
22.4	FTP basics	585
22.4.1	Active mode FTP.....	586
22.4.2	Passive mode FTP.....	587
22.4.3	FTP reply codes	588
22.4.4	Supported FTP commands	589
22.5	Using the FTP server sample	590
22.5.1	Using the Windows sample.....	590
22.5.2	Running the FTP server example on target hardware	590
22.6	Access control	591
22.7	Configuration	597
22.7.1	Compile time configuration switches.....	597
22.7.2	FTP server system time	598
22.8	API functions	600
22.9	FTP server data structures	603
22.10	Resource usage	606
22.10.1	ROM usage on an ARM7 system.....	606
22.10.2	ROM usage on a Cortex-M3 system.....	606
22.10.3	RAM usage:	606
23	FTP client (Add-on).....	607
23.1	embOS/IP FTP client	608
23.2	Feature list	609
23.3	Requirements.....	610
23.4	FTP basics	611
23.4.1	Passive mode FTP.....	612
23.4.2	Supported FTP client commands	613
23.5	Configuration	614
23.5.1	Compile time configuration switches.....	614
23.6	API functions	615
23.7	FTP client data structures	623
23.8	Resource usage	624

23.8.1	ROM usage on an ARM7 system	624
23.8.2	ROM usage on a Cortex-M3 system	624
23.8.3	RAM usage:	624
24	TFTP client/server	625
24.1	embOS/IP TFTP	626
24.2	Feature list	627
24.3	TFTP basics	628
24.4	Using the TFTP samples	629
24.4.1	Running the TFTP server example on target hardware	629
24.5	API functions	630
24.6	Resource usage	635
24.6.1	ROM usage on an ARM7 system	635
24.6.2	ROM usage on a Cortex-M3 system	635
24.6.3	RAM usage:	635
25	PPP / PPPoE (Add-on)	637
25.1	embOS/IP PPP/PPPoE	638
25.2	Feature list	639
25.3	Requirements	640
25.4	PPP backgrounds	641
25.5	API functions	642
25.6	PPPoE functions	643
25.7	PPP functions	649
25.8	Modem functions	655
25.9	PPP data structures	667
25.10	PPPoE resource usage	673
25.10.1	ROM usage on an ARM7 system	673
25.10.2	ROM usage on a Cortex-M3 system	673
25.10.3	RAM usage	673
25.11	PPP resource usage	674
25.11.1	ROM usage on an ARM7 system	674
25.11.2	RAM usage	674
26	NetBIOS (Add-on)	675
26.1	embOS/IP NetBIOS	676
26.2	Feature list	677
26.3	Requirements	678
26.4	NetBIOS backgrounds	679
26.5	API functions	680
26.6	Resource usage	685
26.6.1	ROM usage on an ARM7 system	685
26.6.2	ROM usage on a Cortex-M3 system	685
26.6.3	RAM usage	685
27	SNTP client (Add-on)	687
27.1	embOS/IP SNTP client	688
27.2	Feature list	689
27.3	Requirements	690
27.4	SNTP backgrounds	691
27.4.1	The NTP timestamp	691
27.4.2	The epoch problem (year 2036 problem)	692
27.5	API functions	693
27.6	Resource usage	697
27.6.1	ROM usage on an ARM7 system	697
27.6.2	ROM usage on a Cortex-M3 system	697
27.6.3	RAM usage	697
28	SNMP agent (Add-on)	699

28.1	embOS/IP SNMP agent	700
28.2	Feature list	701
28.3	Requirements.....	702
28.4	SNMP backgrounds	703
28.4.1	Data organization in SNMP	704
28.4.2	OID value, address and index.....	705
28.4.3	SNMP data types	706
28.4.4	Participants in an SNMP environment	708
28.4.5	Differences between SNMP versions	709
28.4.6	SNMP communication basics	710
28.4.7	SNMP agent return codes.....	711
28.5	Using the SNMP agent samples.....	712
28.5.1	OS_IP_SNMP_Agent.c	712
28.5.2	OS_IP_SNMP_Agent_ZeroCopy.c	712
28.5.3	Using the Windows sample.....	713
28.5.4	Features of the sample application	713
28.5.5	Testing the sample application	714
28.6	The MIB callback	716
28.7	Configuration	719
28.7.1	Configuration macro types	719
28.7.2	Compile time configuration switches.....	719
28.8	API functions	720
28.9	SNMP agent data structures	783
28.10	Resource usage	787
28.10.1	ROM usage on an ARM7 system.....	787
28.10.2	ROM usage on a Cortex-M3 system	787
28.10.3	RAM usage:	787
29	Profiling with SystemView	789
29.1	Overview	790
29.2	Additional files	791
29.3	Enable profiling	792
29.4	Recording and analysing profiling information	793
30	Debugging.....	795
30.1	Message output.....	796
30.2	Testing stability	797
30.3	API functions	798
30.4	Message types	804
30.5	Using a network sniffer to analyse communication problems.....	806
31	OS integration	807
31.1	General information	808
31.2	OS layer API functions.....	809
31.2.1	Examples	809
32	Performance & resource usage	811
32.1	Memory footprint.....	812
32.1.1	ARM7 system	812
32.1.2	Cortex-M3 system	813
32.2	Performance	814
32.2.1	ARM7 system	814
32.2.2	Cortex-M3 system	815
33	Appendix A - File system abstraction layer	817
33.1	File system abstraction layer	818
33.2	File system abstraction layer function table	819
33.2.1	emFile interface.....	821
33.2.2	Read-only file system.....	822

33.2.3	Using the read-only file system	823
33.2.4	Windows file system interface	824
34	Support	825
34.1	Contacting support	826
35	Glossary	827

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.

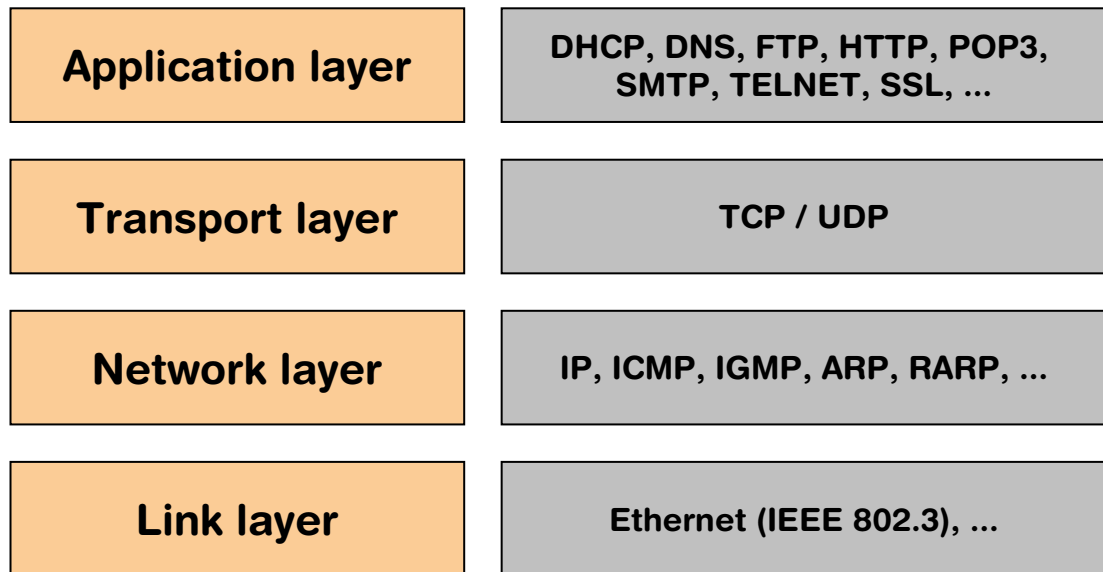
Some features of embOS/IP:

- 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.

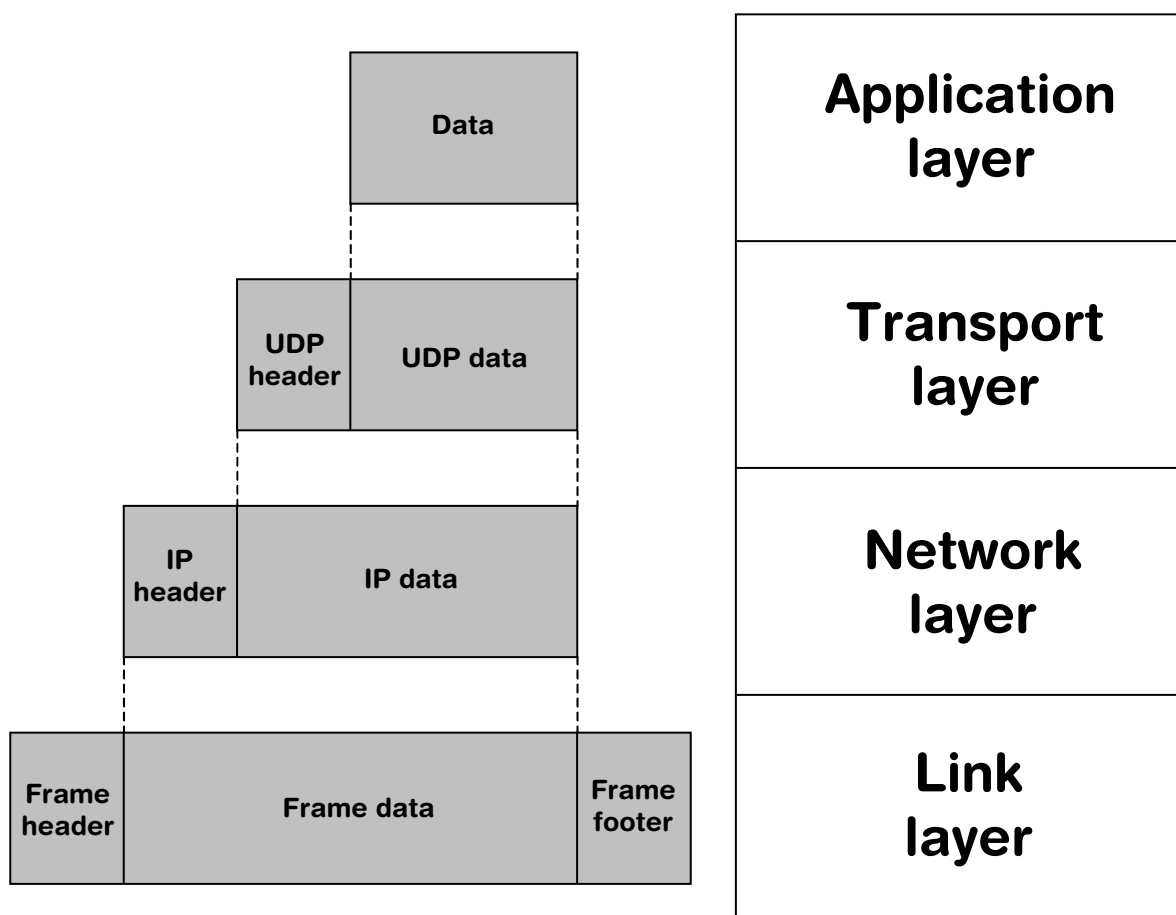
Link layer

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 task `IP_Task` should be higher than the priority of all application tasks that use the stack to allow optimal performance. The `IP_RxTask` (if available) needs to run at the highest single task priority of all IP related task as it is an interrupt moved into a task.

Task priorities

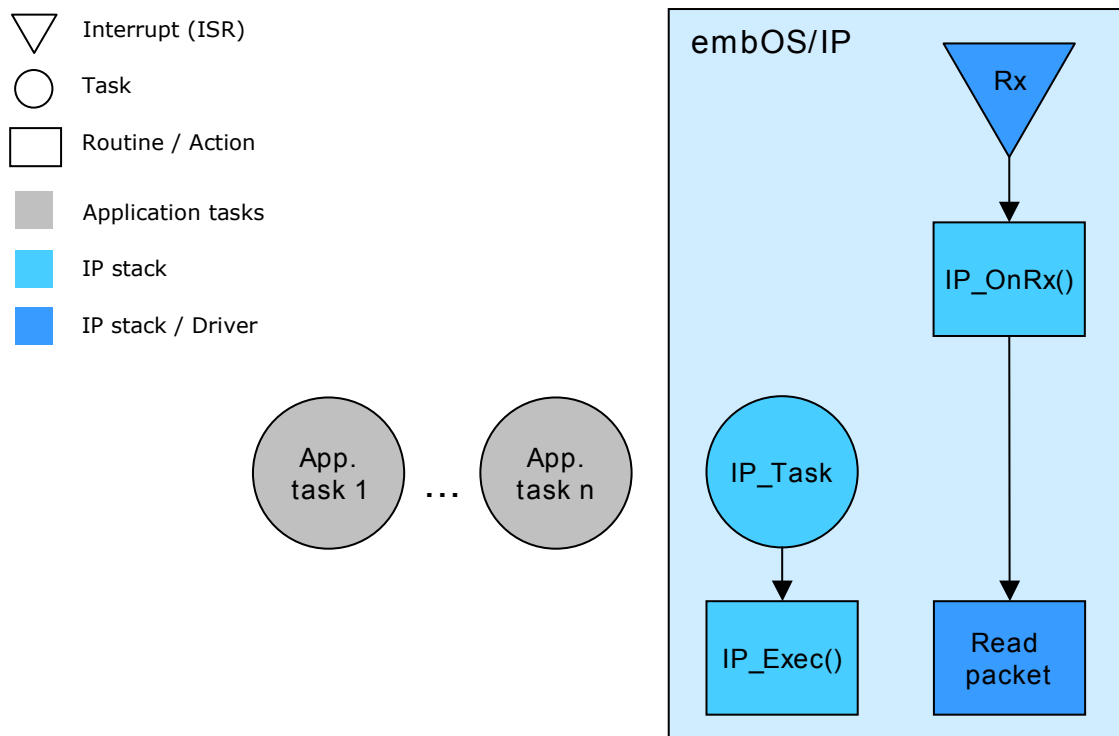
Task priorities are independent from other task priorities. However as soon as a task calls an IP API it should follow these priority rules for best performance of the stack:

1. The `IP_RxTask` (if used at all) needs to have the highest single priority of all tasks that make use of the IP API, having a higher priority than the `IP_Task`.
2. The `IP_Task` should have a higher task priority than any other task that makes use of the IP API. It needs to have a lower priority than the `IP_RxTask` (if used at all).
3. All tasks that make use of the IP API should use a task priority below the `IP_Task` to allow optimal performance.

Task priorities for tasks not using the IP API can be freely chosen.

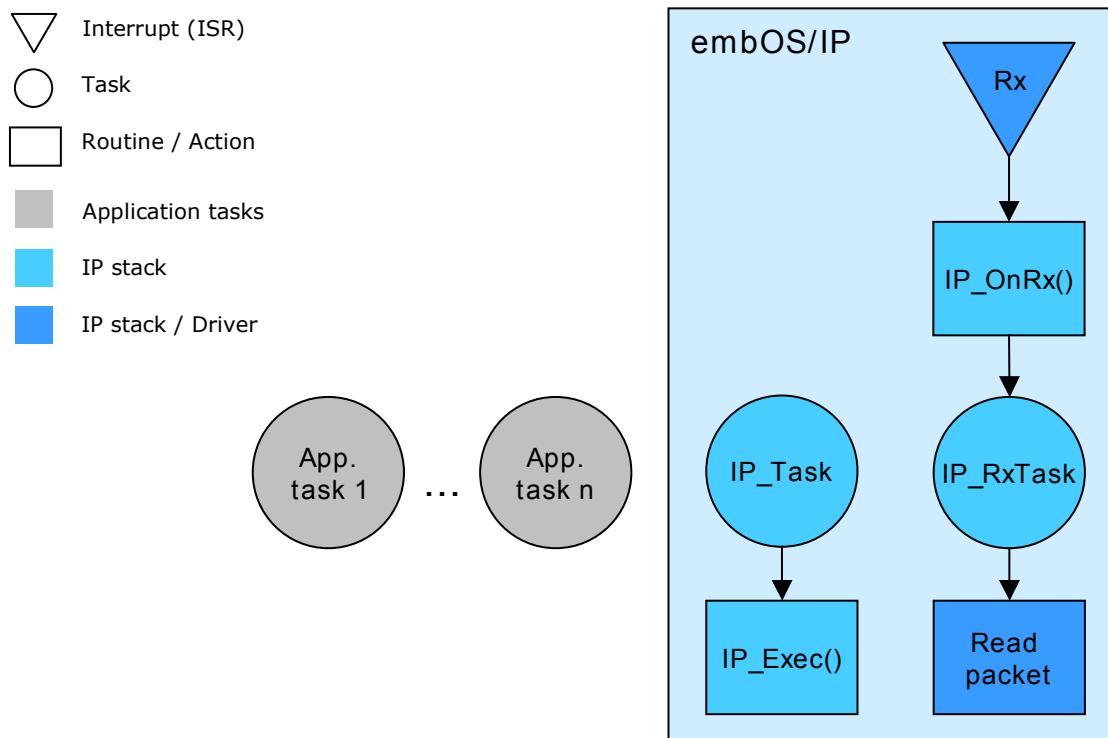
One task dedicated to the stack

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 120 for more information and an example about how to include the `IP_Task` into your embOS project.



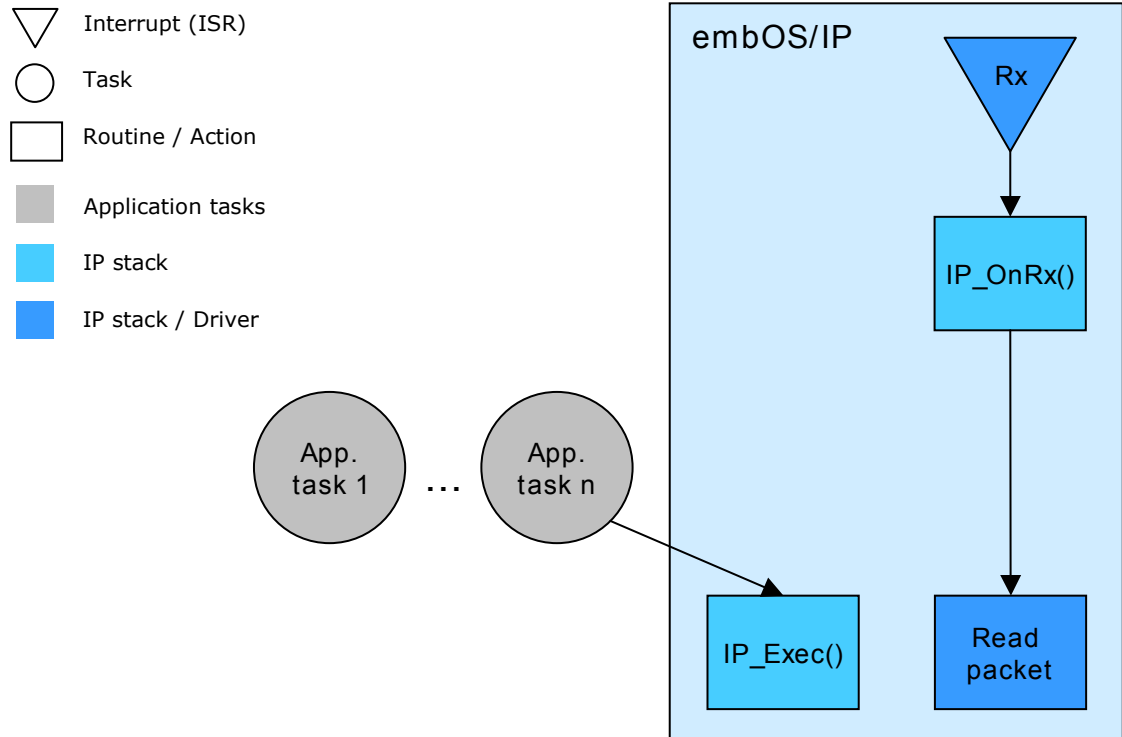
Two tasks dedicated to the stack

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 120 and `IP_RxTask()` on page 121 for more information. `IP_RxTask` has to have a higher priority as `IP_Task` as it is treated as interrupt in task form and might not be interrupted by `IP_Task` or any other IP task.



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.



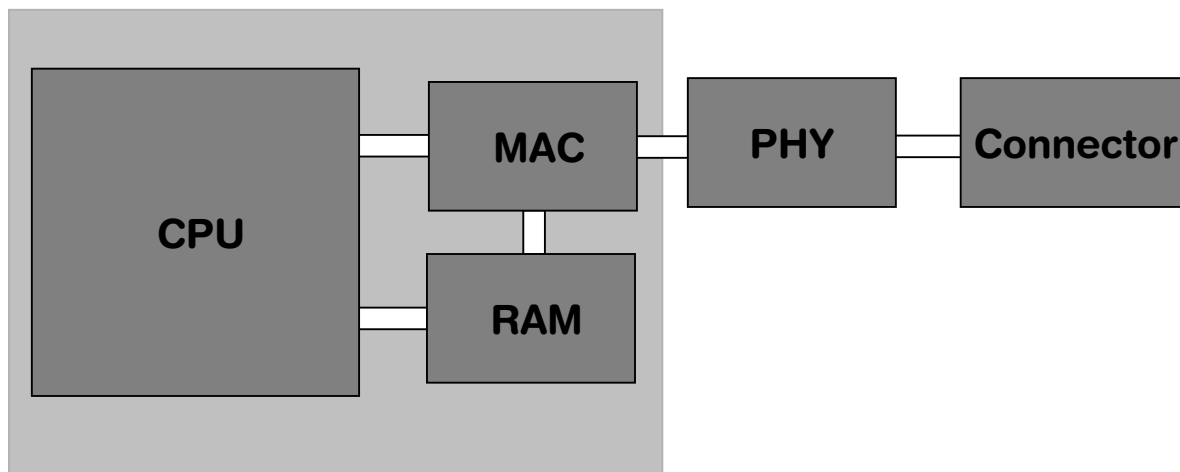
1.5 Background information

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.

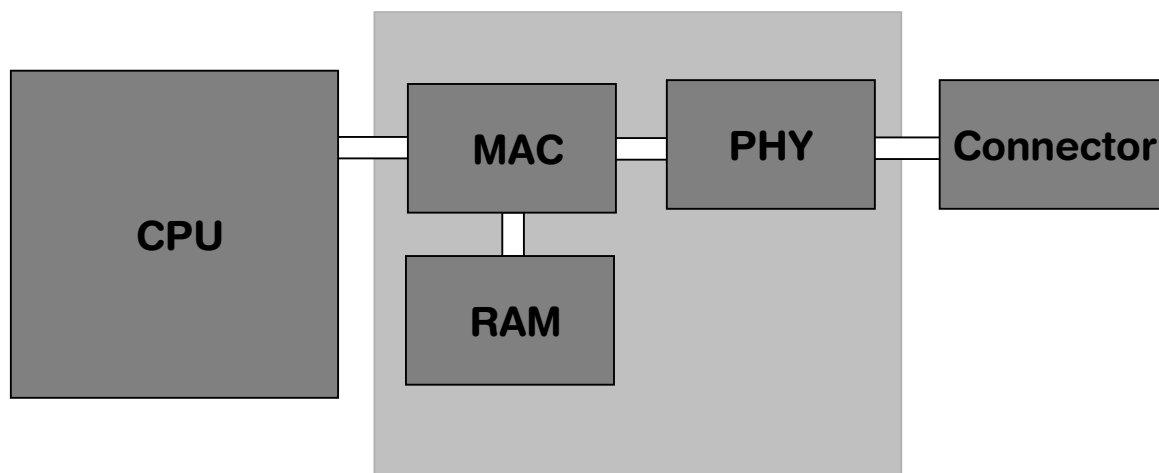
MCUs with integrated MAC

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.

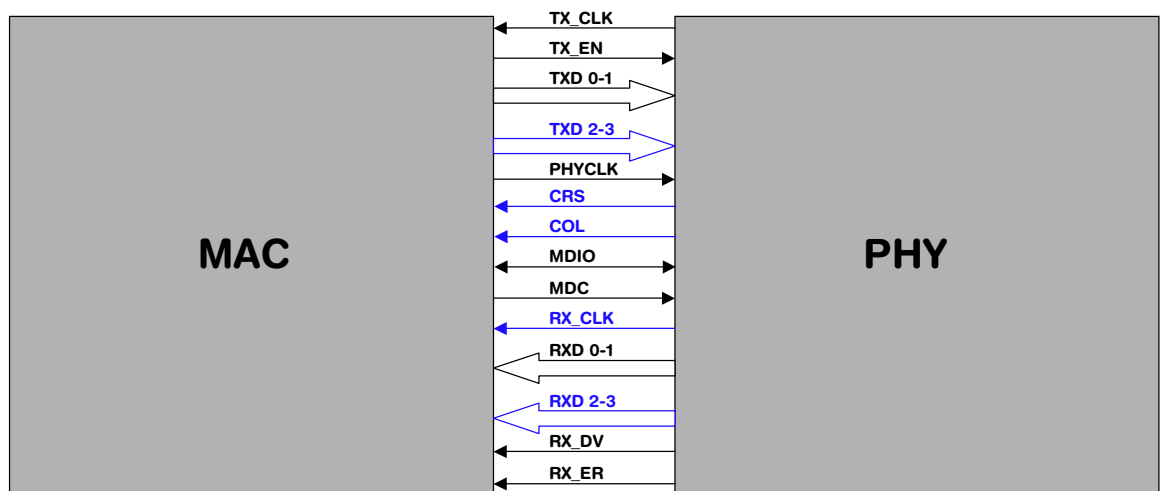
The standard defines 32 16-bit PHY registers. The first 6 are defined by the standard.

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.txt
[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".

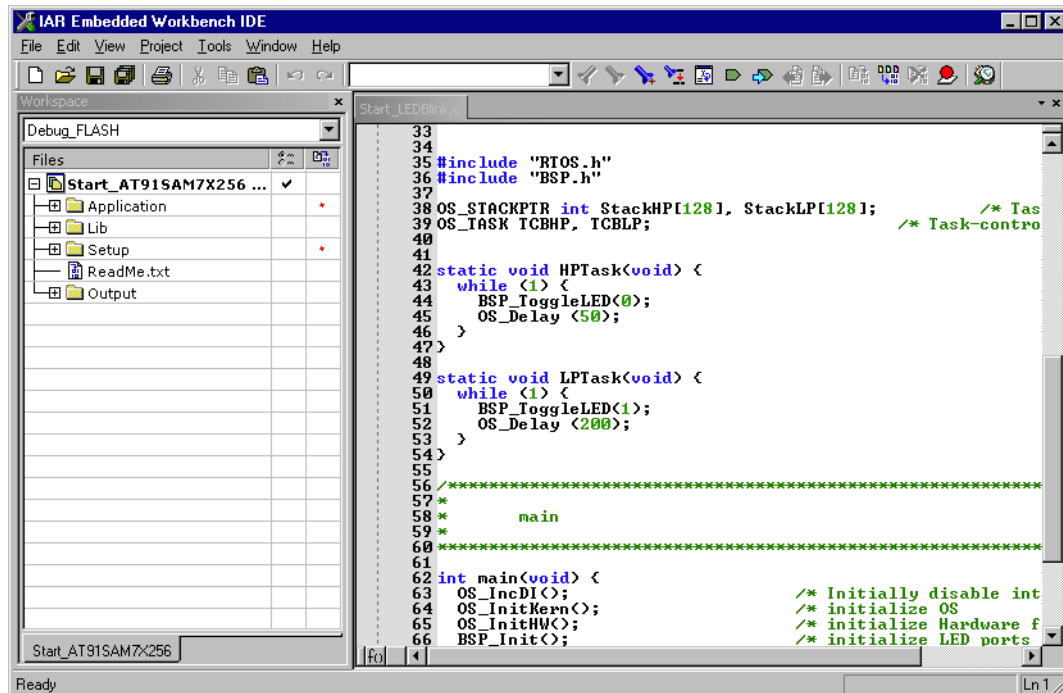
Procedure to follow

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
- IP\ASM (optimized assembler routines)
- 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 `IP\ASM` folder contains files for various CPUs and toolchains with routines optimized in assembler code. Typically only one of these files needs to be added to your project and the rest should be excluded. The optimized routines are used by overwriting a specific macro that typically can be found in `Config\IP_Conf.h`.

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 memcpy 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.

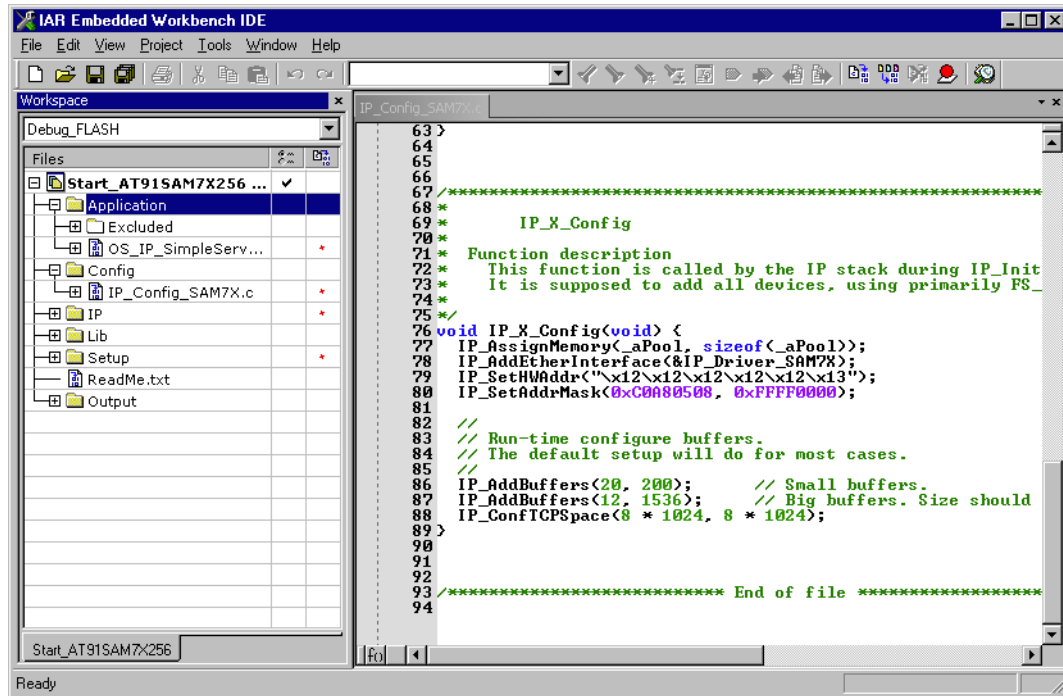
Configuring the include path

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

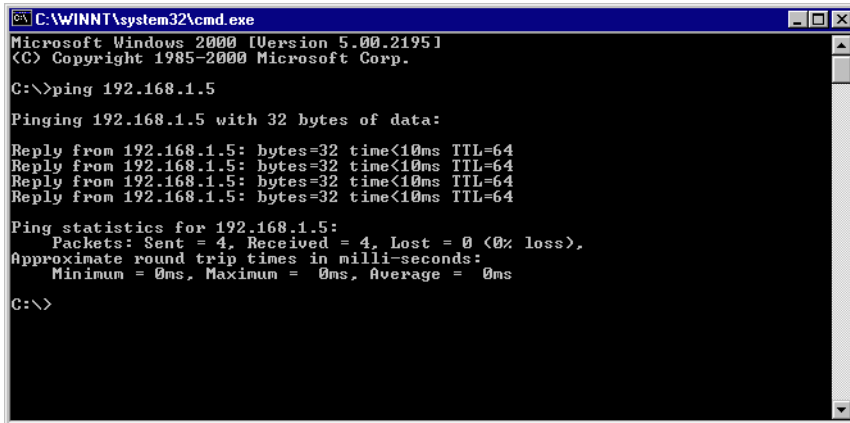
For quick and easy testing of your embOS/IP integration, start with the code found in the folder `Application`. Add one of the applications to your project (for example `OS_IP_SimpleServer.c`).



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.



```
C:\WINNT\system32\cmd.exe
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>ping 192.168.1.5

Pinging 192.168.1.5 with 32 bytes of data:

Reply from 192.168.1.5: bytes=32 time<10ms TTL=64
Reply from 192.168.1.5: bytes=32 time<10ms TTL=64
Reply from 192.168.1.5: bytes=32 time<10ms TTL=64
Reply from 192.168.1.5: bytes=32 time<10ms TTL=64

Ping statistics for 192.168.1.5:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\>
```


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 <i>embOS/IP speed client (OS_IP_SpeedClient_TCP.c)</i> on page 44 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: <code>ping <target-ip></code> where <code><target-ip></code> represents the IP address of the target, which depends on the configuration and is usually <code>192.168.5.1</code> if the DHCP client is not enabled.
OS_IP_UDPDDiscover.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_UDPDDiscoverZeroCopy.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 a 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.

```

C:\WINNT\system32\cmd.exe - telnet 192.168.199.12
IP-Shell.
Enter command (? for help)
List of supported commands
arp --> Show arp status
icmp --> Show ICMP statistics
tcp --> Show TCP statistics
sock --> Show socket list
bsd -->
bsdrecv -->
mbuf --> Show memory buffer statistics
mbl --> Show memory buffer list
stat --> Show MIB statistics
udp -->
udpsock --> List UDP sockets
dhcp --> Show status of DHCP client
dns --> Show status of DNS
dns1 --> Show status of DNS
  
```

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)

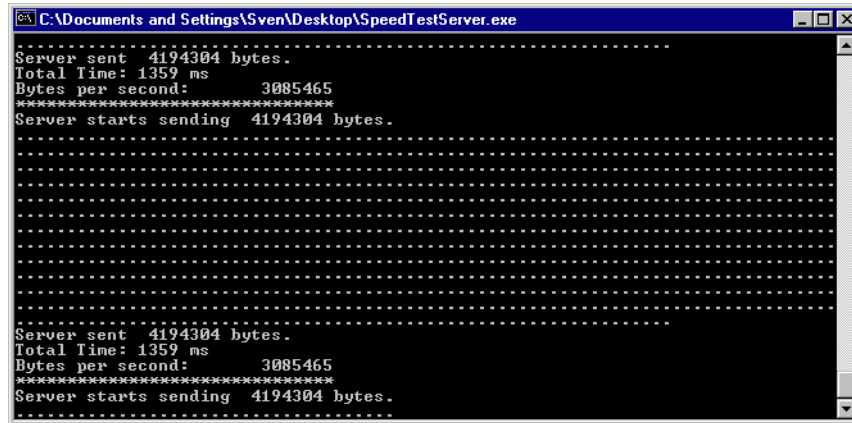
The embOS/IP speed client is a small application to detect the TCP send and receive performance of embOS/IP on your hardware.

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.



```

C:\Documents and Settings\Sven\Desktop\SpeedTestServer.exe
Server sent 4194304 bytes.
Total Time: 1359 ms
Bytes per second: 3085465
*****
Server starts sending 4194304 bytes.

.....

Server sent 4194304 bytes.
Total Time: 1359 ms
Bytes per second: 3085465
*****
Server starts sending 4194304 bytes.

```

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	
<code>IP_AddBuffers()</code>	Adds buffers to the TCP/IP stack.
<code>IP_AddEtherInterface()</code>	Adds an Ethernet interface to the stack.
<code>IP_AddVirtEtherInterface()</code>	Adds a virtual Ethernet interface to the stack.
<code>IP_AddLoopbackInterface()</code>	Adds an Ethernet loopback interface.
<code>IP_AddMemory()</code>	Adds additional memory to the stack.
<code>IP_AllowBackpressure()</code>	Activates back pressure.
<code>IP_AssignMemory()</code>	Assigns memory.
<code>IP_ARP_ConfigAgeout()</code>	Configures the ARP cache timeout.
<code>IP_ARP_ConfigAgeoutNoReply()</code>	Configures the ARP cache timeout for request sent without a reply yet.
<code>IP_ARP_ConfigAgeoutSniff()</code>	Configures the ARP cache timeout for entries sniffed from incoming packets.
<code>IP_ARP_ConfigAllowGratuitousARP()</code>	Configures allow/disallow of using information from gratuitous ARP packets.
<code>IP_ARP_ConfigMaxPending()</code>	Configure max. packets pending for reply to an ARP entry.
<code>IP_ARP_ConfigMaxRetries()</code>	Configures max. ARP request resends.
<code>IP_ARP_ConfigNumEntries()</code>	Configures number of ARP cache entries.
<code>IP_ConfigNumLinkDownProbes()</code>	Configures the number of continuous link down probes to take before the stack accepts the link down status.
<code>IP_ConfigNumLinkUpProbes()</code>	Configures the number of continuous link up probes to take before the stack accepts the link up status.
<code>IP_ConfigOffCached2Uncached()</code>	Configures cached to uncached offset.
<code>IP_ConfigTCPSpace()</code>	Configures the send and receive space.
<code>IP_DisableIPRxChecksum()</code>	Disables IP checksum verification.
<code>IP_CACHE_SetConfig()</code>	Sets a cache configuration to use.
<code>IP_DNS_GetServer()</code>	Retrieves first DNS server from first interface.
<code>IP_DNS_GetServerEx()</code>	Retrieves a DNS server from an interface.
<code>IP_DNS_SetMaxTTL()</code>	Sets the maximum TTL of a DNS entry.
<code>IP_DNS_SetServer()</code>	Sets the DNS server.
<code>IP_DNS_SetServerEx()</code>	Sets a DNS server for an interface.
<code>IP_EnableIPRxChecksum()</code>	Enables ICMP checksum verification.
<code>IP_GetMaxAvailPacketSize()</code>	Retrieves the maximum size of a packet that can be allocated.
<code>IP_GetMTU()</code>	Retrieves the MTU configured for an interface.
<code>IP_GetPrimaryIFace()</code>	Retrieves the stacks primary interface.
<code>IP_ICMP_Add()</code>	Adds ICMP to the stack.

Table 4.1: embOS/IP API function overview

Function	Description
<code>IP_ICMP_DisableRxChecksum()</code>	Disables ICMP checksum verification.
<code>IP_ICMP_EnableRxChecksum()</code>	Enables ICMP checksum verification.
<code>IP_IGMP_Add()</code>	Adds IGMP to the stack.
<code>IP_IGMP_AddEx()</code>	Adds IGMP to the stack.
<code>IP_IGMP_JoinGroup()</code>	Joins an IGMP group.
<code>IP_IGMP_LeaveGroup()</code>	Leaves an IGMP group.
<code>IP_NI_ConfigPoll()</code>	Select polled mode for the network interface.
<code>IP_NI_SetTxBufferSize()</code>	Configures the Tx buffer size used by the network interface driver.
<code>IP_RAW_Add()</code>	Adds RAW socket support to the stack.
<code>IP_SetAddrMask()</code>	Sets the address mask of the first interface interface.
<code>IP_SetAddrMaskEx()</code>	Sets the address mask of the selected interface.
<code>IP_SetGWAddr()</code>	Sets the gateway address of the selected interface.
<code>IP_SetHWAddr()</code>	Sets the hardware address of the first interface.
<code>IP_SetHWAddrEx()</code>	Sets the hardware address of the selected interface.
<code>IP_SetMTU()</code>	Sets the maximum transmission unit of an interface.
<code>IP_SetPrimaryIFace()</code>	Sets primary interface of the stack.
<code>IP_SetSupportedDuplexModes()</code>	Sets the supported duplex modes.
<code>IP_SetTTL()</code>	Sets the TTL of an IP packet.
<code>IP_SOCKET_ConfigSelectMultiplicator()</code>	Configure the select() timeout multiplier.
<code>IP_SOCKET_SetDefaultOptions()</code>	Sets the socket options which should be enabled by default.
<code>IP_SOCKET_SetLimit()</code>	Sets the maximum number of available sockets.
<code>IP_TCP_Add()</code>	Adds TCP to the stack.
<code>IP_TCP_DisableRxChecksum()</code>	Disables TCP checksum verification.
<code>IP_TCP_EnableRxChecksum()</code>	Enables TCP checksum verification.
<code>IP_TCP_Set2MSLDelay()</code>	Sets the maximum segment lifetime.
<code>IP_TCP_SetConnKeepaliveOpt()</code>	Sets the keepalive options.
<code>IP_TCP_SetRetransDelayRange()</code>	Sets retransmission delay range.
<code>IP_UDP_Add()</code>	Adds UDP to the stack.
<code>IP_UDP_AddEchoServer()</code>	Adds a simple echo server.
<code>IP_UDP_DisableRxChecksum()</code>	Disables UDP checksum verification.
<code>IP_UDP_EnableRxChecksum()</code>	Enables UDP checksum verification.
Management functions	
<code>IP_DeInit()</code>	Deinitialization function of the stack.
<code>IP_Init()</code>	Initialization function of the stack.
<code>IP_Task()</code>	Main task for starting the stack.
<code>IP_RxTask()</code>	Reads all available packets and sleeps until a new packet is received.
<code>IP_Exec()</code>	Checks if any packet has been received and handles timers.
Network interface configuration and handling functions	

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

Function	Description
<code>IP_NI_ConfigPoll()</code>	Select polled mode for the network interface.
<code>IP_NI_ForceCaps()</code>	Allows forcing of hardware capabilities.
<code>IP_NI_SetTxBufferSize()</code>	Configures the Tx buffer size used by the network interface driver.
PHY configuration functions	
<code>IP_NI_ConfigPHYAddr()</code>	Configures the PHY address.
<code>IP_NI_ConfigPHYMode()</code>	Configures the PHY mode.
<code>IP_PHY_AddDriver()</code>	Adds a PHY driver and assigns it to an interface.
<code>IP_PHY_AddResetHook()</code>	Adds a hook that will be executed after a PHY software reset has been applied by the stack.
<code>IP_PHY_ConfigAddr()</code>	Configures the PHY addr. to use.
<code>IP_PHY_ConfigAltAddr()</code>	Configure alternate PHY addresses for use with switches.
<code>IP_PHY_ConfigSupportedModes()</code>	Configures the duplex and speed modes that shall be supported.
<code>IP_PHY_ConfigUseStaticFilters()</code>	Configures if using PHY filter is allowed.
<code>IP_PHY_DisableCheck()</code>	Disables PHY checks for all interfaces.
<code>IP_PHY_DisableCheckEx()</code>	Disables PHY checks for one interface.
<code>IP_PHY_ReInit()</code>	Re-initializes the PHY.
<code>IP_PHY_SetWdTimeout()</code>	Sets the PHY watchdog timeout.
Statistics functions	
<code>IP_STATS_EnableIFaceCounters()</code>	Enables statistic counters for a specific interface.
<code>IP_STATS_GetIFaceCounters()</code>	Retrieves a pointer to the statistic counters for a specific interface.
<code>IP_STATS_GetLastLinkStateChange()</code>	Retrieves the tick count when an interface entered its current state.
<code>IP_STATS_GetRxBytesCnt()</code>	Retrieves the number of bytes received on an interface.
<code>IP_STATS_GetRxDiscardCnt()</code>	Retrieves the number of packets received but discarded although they were O.K. .
<code>IP_STATS_GetRxErrCnt()</code>	Retrieves the number of receive errors.
<code>IP_STATS_GetRxNotUnicastCnt()</code>	Retrieves the number of packets received on an interface that were not unicasts.
<code>IP_STATS_GetRxUnicastCnt()</code>	Retrieves the number of unicast packets received on an interface.
<code>IP_STATS_GetRxUnknownProtoCnt()</code>	Retrieves the number of unknown protocols received.
<code>IP_STATS_GetTxBytesCnt()</code>	Retrieves the number of bytes sent on an interface.
<code>IP_STATS_GetTxDiscardCnt()</code>	Retrieves the number of packets to send but discarded although they were O.K. .
<code>IP_STATS_GetTxErrCnt()</code>	Retrieves the number of send errors on an interface.

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

Function	Description
<code>IP_STATS_GetTxNotUnicastCnt()</code>	Retrieves the number of packets sent on an interface that were not unicasts.
<code>IP_STATS_GetTxUnicastCnt()</code>	Retrieves the number of unicast packets sent on an interface.
Other IP stack functions	
<code>IP_AddAfterInitHook()</code>	Adds a hook that will be executed right after <code>IP_Init()</code> .
<code>IP_AddEtherTypeHook()</code>	This function registers a callback to be called for received packets with the registered Ethernet type.
<code>IP_AddOnPacketFreeHook()</code>	This function adds a hook function to the <code>IP_HOOK_ON_PACKET_FREE</code> list. Registered hooks will be called in case a packet gets freed.
<code>IP_AddStateChangeHook()</code>	Adds a hook that will be executed if an interface state changes from outside.
<code>IP_Alloc()</code>	Thread safe memory allocation from main IP stack memory pool.
<code>IP_AllocEtherPacket()</code>	Allocates a packet to store the raw data of an Ethernet packet of up to <code>NumBytes</code> at the location returned by <code>ppBuffer</code> .
<code>IP_AllocEx()</code>	Thread safe memory allocation from a specific memory pool managed by the stack that has been added using <code>IP_AddMemory()</code> .
<code>IP_ARP_CleanCache()</code>	Cleans the ARP cache.
<code>IP_ARP_CleanCacheByInterface()</code>	Cleans the ARP cache by interface.
<code>IP_Connect()</code>	Calls a previously set callback.
<code>IP_Disconnect()</code>	Calls a previously set callback.
<code>IP_Err2Str()</code>	Converts error value to string.
<code>IP_FindIFaceByIP()</code>	Tries to find out the interface number when only the IP address is known.
<code>IP_Free()</code>	Free previously allocated memory.
<code>IP_FreePacket()</code>	Frees a packet back to the stack.
<code>IP_GetAddrMask()</code>	Returns the IP address and the subnet mask of the device.
<code>IP_GetCurrentLinkSpeed()</code>	Returns the current link speed.
<code>IP_GetCurrentLinkSpeedEx()</code>	Returns the current link speed of the selected interface.
<code>IP_GetFreePacketCnt()</code>	Retrieves the number of free packets.
<code>IP_GetIFaceHeaderSize()</code>	Retrieves the interface header size.
<code>IP_GetGWAddr()</code>	Returns the gateway address of the device.
<code>IP_GetHWAddr()</code>	Returns the hardware address (MAC) of the device.
<code>IP_GetIPAddr()</code>	Returns the IP address of the device.
<code>IP_GetIPPacketInfo()</code>	Returns the start address of the data part of an IP packet.
<code>IP_GetRawPacketInfo()</code>	Returns the start address of the raw data part of an IP packet.
<code>IP_GetVersion()</code>	Returns the version number of embOS/IP.

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

Function	Description
<code>IP_ICMP_SetRxHook()</code>	Sets a hook function which will be called if target receives a ping packet.
<code>IP_IFaceIsReady()</code>	Checks if the interface is ready.
<code>IP_IFaceIsReadyEx()</code>	Checks if the specified interface is ready.
<code>IP_IsExpired()</code>	Checks if a timestamp has expired.
<code>IP_NI_GetAdminState()</code>	Retrieves the admin state of the given interface.
<code>IP_NI_GetIFaceType()</code>	Retrieves a short textual description of the interface type.
<code>IP_NI_GetState()</code>	Returns the hardware state of the interface.
<code>IP_NI_SetAdminState()</code>	Sets the AdminState of the interface.
<code>IP_NI_GetTxQueueLen()</code>	Retrieves the current length of the Tx queue of an interface.
<code>IP_PrintIPAddr()</code>	Convert an 4 byte IP address to a dots-and-number string.
<code>IP_ResolveHost()</code>	Resolves a host name via DNS server.
<code>IP_SendEtherPacket()</code>	Sends a previously allocated Ethernet packet.
<code>IP_SendPacket()</code>	Sends a user defined packet on the interface.
<code>IP_SendPing()</code>	Sends an ICMP Echo Request.
<code>IP_SendPingEx()</code>	Sends an ICMP Echo Request.
<code>IP_SetIFaceConnectHook()</code>	Sets a connect callback.
<code>IP_SetIFaceDisconnectHook()</code>	Sets a disconnect callback.
<code>IP_SetPacketToS()</code>	Sets the ToS byte in the IP header of a packet to be sent.
<code>IP_SetRxHook()</code>	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
<code>NumBuffers</code>	[IN] The number of buffers.
<code>BytesPerBuffer</code>	[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

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

Parameter

Parameter	Description
<code>pDriver</code>	[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 373 for a list of available network interface drivers.

Return value

Zero-based interface index of the newly created interface.

Example

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

4.2.3 IP_AddVirtEtherInterface()

Description

Adds a virtual Ethernet interface.

Prototype

```
int IP_AddVirtEtherInterface( unsigned HWIFaceId );
```

Parameter

Parameter	Description
HWIFaceId	Zero-based interface index of the hardware interface to be used for communication.

Table 4.4: IP_AddVirtEtherInterface() parameter list

Additional information

Virtual interfaces can be added to allow configuration of multiple IP addresses on the same target. One configuration can be assigned per interface.

Return value

Zero-based interface index of the newly created interface.

Example

```
int IFaceId;

IFaceId = IP_AddEtherInterface(&IP_Driver_SAM7X); // Add HW Ethernet driver
IP_AddVirtEtherInterface(IFaceId);
```

4.2.4 IP_AddLoopbackInterface()

Description

Adds a loopback Ethernet interface.

Prototype

```
int IP_AddLoopbackInterface( void );
```

Return value

Zero-based interface index of the newly created interface.

Additional information

The loopback interface will be added with the pre-configured static IP addr. of 127.0.0.1/8 .

Example

```
IP_AddLoopbackInterface(); // Add an Ethernet loopback interface.
```


4.2.5 IP_AddMemory()

Description

Adds additional memory to the TCP/IP stack. Can be called after *IP_AssignMemory()* on page 59.

Prototype

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

Parameter

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

Table 4.5: IP_AddMemory() parameter list

Additional information

This function can be used to add additional memory to the stack that can then be requested by application level modules such as Web server or FTP server directly from the stacks memory management. For further information about the available memory management functions, refer to *IP_Alloc()* on page 162 and *IP_Free()* on page 169.

Example

```
#define MEM_SIZE 0x8000 // Size of memory to add to the stack in bytes.
U32 _aMem[MEM_SIZE / 4]; // Memory area to add to the stack.

IP_AddMemory(_aMem, sizeof(_aMem));
```

4.2.6 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 4.6: 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.

4.2.7 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 4.7: 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));
```

4.2.8 IP_ARP_CleanCache()

Description

Cleans all ARP entries that are not static entries.

Prototype

```
void IP_ARP_CleanCache ( void );
```

4.2.9 IP_ARP_CleanCacheByInterface()

Description

Cleans all ARP entries that are known to belong to a specific interface and are not static entries.

Prototype

```
void IP_ARP_CleanCacheByInterface ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.8: IP_ARP_CleanCacheByInterface() parameter list

4.2.10 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 4.9: IP_ARP_ConfigAgeout() parameter list

4.2.11 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 4.10: IP_ARP_ConfigAgeoutNoReply() parameter list

4.2.12 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 4.11: IP_ARP_ConfigAgeoutSniff() parameter list

4.2.13 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 4.12: 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.

4.2.14 IP_ARP_ConfigMaxPending()

Description

Configures the maximum number packets that can be queued waiting for an ARP reply.

Prototype

```
void IP_ARP_ConfigMaxPending ( unsigned NumPackets );
```

Parameter

Parameter	Description
NumPackets	[IN] Maximum number of packets that can be pending for one ARP entry. Default: 3.

Table 4.13: IP_ARP_ConfigMaxPending() parameter list

4.2.15 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 4.14: IP_ARP_ConfigMaxRetries() parameter list

4.2.16 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 4.15: IP_ARP_ConfigNumEntries() parameter list

Return 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()`.

4.2.17 IP_BSP_SetAPI()

Description

Sets an API to be used for BSP related abstraction like initializing hardware and installing interrupt handlers.

Prototype

```
void IP_BSP_SetAPI(          unsigned    IFaceId,
                          const IP_BSP_API* pAPI );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
pAPI	Pointer to function table to use. For further information regarding IP_BSP_API please refer to <i>Structure IP_BSP_API</i> on page 203.

Table 4.16: IP_BSP_SetAPI() parameter list

4.2.18 IP_CACHE_SetConfig()

Description

Configures cache related functionality that might be required by the stack for several purposes such as cache handling in drivers.

Prototype

```
void IP_CACHE_SetConfig ( const SEGGER_CACHE_CONFIG *pConfig,  
                          unsigned                  ConfSize);
```

Parameter

Parameter	Description
<code>pConfig</code>	[IN] Pointer to an element of SEGGER_CACHE_CONFIG . For further information please refer to <i>Structure SEGGER_CACHE_CONFIG</i> on page 204.
<code>ConfSize</code>	[IN] Size of the passed structure.

Table 4.17: IP_CACHE_SetConfig() parameter list

Additional information

`IP_CACHE_SetConfig()` has to be called before `IP_Init()` or during `IP_X_Config()`. Typically used together with `IP_ConfigOffCached2Uncached()` on page 74.

4.2.19 IP_ConfigMaxIFaces()

Description

Configures the maximum number of interfaces that can be added to the system.

Prototype

```
void IP_ConfigMaxIFaces( unsigned NumIFaces );
```

Parameter

Parameter	Description
NumIFaces	Number of interfaces to allocate memory for.

Table 4.18: IP_ConfigMaxIFaces() parameter list

Additional information

The memory for the driver list will be pre-allocated for the maximum allowed number of interfaces. The system uses the default value of IP_MAX_IFACES if not configured else with this function. To save some memory the maximum number of interfaces should be only the number of interfaces that are really required.

4.2.20 IP_ConfigNumLinkDownProbes()

Description

Configures the number of continuous link down probes to take before the stack accepts the link down status.

Prototype

```
void IP_ConfigNumLinkDownProbes ( U8 IFaceId,  
                                   U8 NumProbes );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
NumProbes	Number of continuous link down probes to take before link down is set in the stack.

Table 4.19: IP_ConfigNumLinkDownProbes() parameter list

Additional information

On unstable hardware or unstable network hardware like a switch a link jitter might occur. This jitter might lead to disconnects on upper protocol layers like TCP that might be disconnected once a link down is recognized. To prevent this to happen due to link jitter, multiple samples of a link down state can be taken before actually accepting the link down.

Typically the link status is checked once per second. Therefore by default [NumProbes](#) = seconds after which the link state in the stack is allowed to really get down after the first link down reported by the driver.

This routine is only effective in case the define `IP_NUM_LINK_DOWN_PROBES` is not 0.

4.2.21 IP_ConfigNumLinkUpProbes()

Description

Configures the number of continuous link up probes to take before the stack accepts the link up status.

Prototype

```
void IP_ConfigNumLinkUpProbes ( U8 IFaceId,
                                U8 NumProbes );
```

Parameter

Parameter	Description
<code>IFaceId</code>	Zero-based interface index.
<code>NumProbes</code>	Number of continuous link up probes to take before link up is set in the stack.

Table 4.20: IP_ConfigNumLinkUpProbes() parameter list

Additional information

Some switches might already report a link between switch and target but are not immediately operational resulting in packets getting lost until fully operational.

Typically the link status is checked once per second. Therefore by default `NumProbes` = seconds after which the link state in the stack is allowed to really get up after the first link up reported by the driver.

At the moment this only applies to Ethernet interfaces to address this behavior with some Ethernet switches.

This routine is only effective in case the define `IP_NUM_LINK_UP_PROBES` is not 0.

4.2.22 IP_ConfigOffCached2Uncached()

Description

Configures the offset from a cached memory area to its uncached equivalent for uncached access.

Prototype

```
void IP_ConfigOffCached2Uncached ( I32 Off );
```

Parameter

Parameter	Description
<code>Off</code>	[IN] Offset from cached to uncached area. Can be negative if uncached area is before cached area.

Table 4.21: IP_ConfigOffCached2Uncached() parameter list

Additional information

This function needs to be called in case the microcontroller is utilizing cache. Typically the data area that is used by default is accessed cached. In this case the stack needs to know where it can bypass the cache to write hardware related data such as driver descriptors that will be accessed by a DMA.

4.2.23 IP_ConfigTCPSpace()

Description

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

Prototype

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

Parameter

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

Table 4.22: 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.

4.2.24 IP_DisableIPRxChecksum()

Description

Disables checksum verification of the checksum in the IP header for incoming packets.

Prototype

```
void IP_DisableIPRxChecksum ( U8 IFace );
```

Parameter

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

Table 4.23: IP_DisableIPRxChecksum() parameter list

Additional information

In a typical network typically all data contained in a transferred frame has already been verified by the hardware checking the transmitted frames checksum and it is unlikely that data within this frame is corrupted if the frame checksum was verified as being correct. Therefore for optimization reasons the checksum calculation might be disabled.

4.2.25 IP_DNS_GetServer()

Description

Retrieves the first DNS server configured of the first interface.

Prototype

```
U32 IP_DNS_GetServer ( void );
```

Return value

First DNS server address of first interface.

4.2.26 IP_DNS_GetServerEx()

Description

Retrieves a DNS server configured for an interface.

Prototype

```
void IP_DNS_GetServerEx ( U8    IFace,  
                          U8    DNSIndex,  
                          U8    *pAddr,  
                          int    *pAddrLen );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based index of available network interfaces. -1 when out of range.
DNSIndex	[IN] Zero-based index of the server to retrieve from interface. -1 when out of range.
pAddr	[OUT] Pointer to U32 variable (for IPv4) to store the DNS addr.
pAddrLen	[OUT] Length of DNS addr. in bytes. Typically 4 for IPv4.

Table 4.24: IP_DNS_GetServerEx() parameter list

4.2.27 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 4.25: 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.

4.2.28 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 4.26: 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 215 for detailed information about resolving an internet address.

4.2.29 IP_DNS_SetServerEx()

Description

Sets one DNS server for an interface.

Prototype

```
void IP_DNS_SetServerEx (      U8    IFace,
                               U8    DNSIndex,
                               const U8 *pDNSAddr,
                               int    AddrLen );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based interface index.
DNSIndex	[IN] Zero-based DNS server index of the interface.
pDNSAddr	[IN] Pointer to memory location holding the DNS addr. to set. Typically an 4-byte IP addr.
AddrLen	[IN] Length of IP addr. of server. Typically 4-bytes.

Table 4.27: IP_DNS_SetServerEx() parameter list

Additional information

If a DHCP server is used for configuring your target, `IP_DNS_SetServerEx()` 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 215 for detailed information about resolving an internet address.

4.2.30 IP_EnableIPRxChecksum()

Description

Enables checksum verification of the checksum in the IP header for incoming packets.

Prototype

```
void IP_EnableIPRxChecksum ( U8 IFace );
```

Parameter

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

Table 4.28: IP_EnableIPRxChecksum() parameter list

Additional information

In a typical network typically all data contained in a transferred frame has already been verified by the hardware checking the transmitted frames checksum and it is unlikely that data within this frame is corrupted if the frame checksum was verified as being correct. Therefore for optimization reasons the checksum calculation might be disabled.

4.2.31 IP_GetMaxAvailPacketSize()

Description

Retrieves the packet size with maximum number of bytes to be allocated that is free in the system.

Prototype

```
U32 IP_GetMaxAvailPacketSize( int IFaceId )
```

Parameter

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

Table 4.29: IP_GetMaxAvailPacketSize() parameter list

Additional information

The packet size returned does not contain any protocol headers other than the transport layer (for Ethernet typically 14 bytes/for PPP typically 6 bytes). Other protocol header such as IPvX and UDPvX need to be subtracted from the value returned.

Return value

0 if no free packet is available at all, otherwise the maximim size available for allocation thorough packet alloc functions such as *IP_UDP_Alloc()* on page 258.

4.2.32 IP_GetMTU()

Description

Retrieves the MTU configured for an interface.

Prototype

```
U32 IP_TCP_GetMTU( U8 IFace )
```

Parameter

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

Table 4.30: IP_GetMTU() parameter list

Return value

MTU configured for the interface, either set as default when adding the interface or set via *IP_SetMTU()* on page 99.

4.2.33 IP_GetPrimaryIFace()

Description

Retrieves the currently set primary interface index of the system.

Prototype

```
int IP_GetPrimaryIFace( void );
```

Return value

Primary interface index set in the system. If not previously configured with *IP_SetPrimaryIFace()* on page 100 the default is 0.

4.2.34 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.

4.2.35 IP_ICMP_DisableRxChecksum()

Description

Disables checksum verification of the checksum in the ICMP header for incoming packets.

Prototype

```
void IP_ICMP_DisableRxChecksum ( U8 IFace );
```

Parameter

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

Table 4.31: IP_ICMP_DisableRxChecksum() parameter list

Additional information

In a typical network typically all data contained in a transferred frame has already been verified by the hardware checking the transmitted frames checksum and it is unlikely that data within this frame is corrupted if the frame checksum was verified as being correct. Therefore for optimization reasons the checksum calculation might be disabled.

4.2.36 IP_ICMP_EnableRxChecksum()

Description

Enables checksum verification of the checksum in the ICMP header for incoming packets.

Prototype

```
void IP_ICMP_EnableRxChecksum ( U8 IFace );
```

Parameter

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

Table 4.32: IP_ICMP_EnableRxChecksum() parameter list

Additional information

In a typical network typically all data contained in a transferred frame has already been verified by the hardware checking the transmitted frames checksum and it is unlikely that data within this frame is corrupted if the frame checksum was verified as being correct. Therefore for optimization reasons the checksum calculation might be disabled.

4.2.37 IP_IGMP_Add()

Description

Adds IGMP to the stack.

Prototype

```
int 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.

For interfaces other than #0 use `IP_IGMP_AddEx()` on page 90 instead.

Return value

`== 0`: O.K.

`!= 0`: Error.

4.2.38 IP_IGMP_AddEx()

Description

Adds IGMP to the stack.

Prototype

```
int IP_IGMP_Add ( unsigned IFaceId );
```

Additional information

Same as *IP_IGMP_Add()* on page 89 but can be added for a specific interface instead of interface #0 only.

Return value

== 0: O.K.

!= 0: Error.

4.2.39 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 4.33: IP_IGMP_JoinGroup() parameter list

Additional information

Calling this function should be only done after `IP_init()` as we rely 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 duplicate 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 queries 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 0xEFFFFFFFA // 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);
```

4.2.40 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 4.34: 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 queries 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  0xEFFFFFFFA  // 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);
```

4.2.41 IP_RAW_Add()

Description

Adds RAW socket support to the stack.

Prototype

```
void IP_RAW_Add ( void );
```

Additional information

`IP_RAW_Add()` adds RAW socket support to the stack. The function should be called during the initialization of the stack.

4.2.42 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 4.35: 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 291 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
```

4.2.43 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 4.36: 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 291 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
```

4.2.44 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 4.37: 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 291 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
```


4.2.45 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 4.38: 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" );
```

4.2.46 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 4.39: 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");
```

4.2.47 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 4.40: 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 421.

4.2.48 IP_SetPrimaryIFace()

Description

Allows to set the primary interface index of the system

Prototype

```
int IP_SetPrimaryIFace( int IFaceId );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index to use as primary interface of the system.

Table 4.41: IP_SetPrimaryIFace() parameter list

Return value

0: O.K.
< 0: Error, interface index might not be valid.

Additional information

The primary interface will be handled with priority in several situations e.g. finding a suitable DNS server to resolve a host name.

4.2.49 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 4.42: 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
IP_PHY_MODE_1000_HALF	Support 1000 Mbit half-duplex
IP_PHY_MODE_1000_FULL	Support 1000 Mbit full-duplex

4.2.50 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 4.43: 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.

4.2.51 IP_SOCKET_ConfigSelectMultiplier()

Description

Allows to configure the multiplier for the timeout parameter of *select()* on page 226.

Prototype

```
void IP_SOCKET_ConfigSelectMultiplier( U32 v );
```

Parameter

Parameter	Description
v	[IN] Multiplier to be used. Default 1.

Table 4.44: IP_SOCKET_ConfigSelectMultiplier() parameter list

Additional information

By default the *select()* timeout is given in ticks of 1 ms. The UNIX standard takes the timeout in a structue including seconds. The multiplier can be configured but as it is more common for an embedded system we will stick to units of 1 tick (typically 1 ms) for the default.

4.2.52 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_KEEPAIVE) socket option is enabled. Refer to <i>setsockopt()</i> on page 231 for a list of supported socket options.

Table 4.45: IP_SOCKET_SetDefaultOptions() parameter list

4.2.53 IP_SOCKET_SetLimit()

Description

Sets the maximum number of available sockets.

Prototype

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

Parameter

Parameter	Description
<code>Limit</code>	[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 4.46: IP_SOCKET_SetLimit() parameter list

4.2.54 IP_SYSVIEW_Init()

Description

Initializes the embOS/IP profile instrumentation and SystemView as profiling implementation.

Prototype

```
void IP_SYSVIEW_Init( void );
```

Additional information

For further information regarding the SysView profiling implementation in embOS/IP please refer to the chapter *Profiling with SystemView* on page 789.

4.2.55 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.

4.2.56 IP_TCP_DisableRxChecksum()

Description

Disables checksum verification of the checksum in the TCP header for incoming packets.

Prototype

```
void IP_TCP_DisableRxChecksum ( U8 IFace );
```

Parameter

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

Table 4.47: IP_TCP_DisableRxChecksum() parameter list

Additional information

In a typical network typically all data contained in a transferred frame has already been verified by the hardware checking the transmitted frames checksum and it is unlikely that data within this frame is corrupted if the frame checksum was verified as being correct. Therefore for optimization reasons the checksum calculation might be disabled.

4.2.57 IP_TCP_EnableRxChecksum()

Description

Enables checksum verification of the checksum in the TCP header for incoming packets.

Prototype

```
void IP_TCP_EnableRxChecksum ( U8 IFace );
```

Parameter

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

Table 4.48: IP_TCP_EnableRxChecksum() parameter list

Additional information

In a typical network typically all data contained in a transferred frame has already been verified by the hardware checking the transmitted frames checksum and it is unlikely that data within this frame is corrupted if the frame checksum was verified as being correct. Therefore for optimization reasons the checksum calculation might be disabled.

4.2.58 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 4.49: 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.

4.2.59 IP_TCP_SetConnKeepaliveOpt()

Description

Sets the keepalive options.

Prototype

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

Parameter

Parameter	Description
<code>Init</code>	Maximum time after TCP-connection open (response to SYN) in ms in case no data transfer takes place. The embOS/IP default is 10 seconds.
<code>Idle</code>	Time of TCP-inactivity before first keepalive probe is sent in ms. The embOS/IP default is 10 seconds.
<code>Period</code>	Time of TCP-inactivity between keepalive probes in ms. The embOS/IP default is 10 seconds.
<code>MaxRep</code>	Number of keepalive probes before we give up and close the connection. The embOS/IP default is 8 repetitions.

Table 4.50: 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.

Keepalives will be sent if the TCP connection sits idle for `Idle` ms and will then start sending a keepalive each `Period` ms for `MaxRep`. Each time a keepalive is ACKed by the peer, the next keepalive will again be sent after `Idle` ms.

By design keepalives are retransmissions of already sent and ACKed data. Depending on the used IP stack a retransmit is typically one byte sent with the current sequence number - 1, so that the peer will discard the data itself as it has already been received and ACKed but will send an ACK back to notify the sender that it has been received and to not send it again.

Other stacks might even send a TCP packet with zero data and the current sequence number, forcing the other side to practically answer back to a duplicate ACK.

Keepalives might not be displayed correctly by tools like Wireshark. A zero length keepalive is typically seen like a duplicate ACK while an one byte keepalive might actually be an one byte retransmit if sending chunks of one byte and one of them has not been ACKed.

The `Init` value configured is the connect timeout that will be used for `connect()` on page 213.

4.2.60 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 4.51: 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.

4.2.61 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.

4.2.62 IP_UDP_AddEchoServer()

Description

Adds a simple echo server for UDP packets that can be used for UDP pings and other tests.

Prototype

```
IP_UDP_CONNECTION * IP_UDP_AddEchoServer ( U16 LPort );
```

Parameter

Parameter	Description
LPort	[IN] Port to listen on for incoming UDP packets.

Table 4.52: IP_UDP_AddEchoServer() parameter list

Additional information

The echo server will simply send back the incoming packet to the sender.

4.2.63 IP_UDP_DisableRxChecksum()

Description

Disables checksum verification of the checksum in the UDP header for incoming packets.

Prototype

```
void IP_UDP_DisableRxChecksum ( U8 IFace );
```

Parameter

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

Table 4.53: IP_UDP_DisableRxChecksum() parameter list

Additional information

In a typical network typically all data contained in a transferred frame has already been verified by the hardware checking the transmitted frames checksum and it is unlikely that data within this frame is corrupted if the frame checksum was verified as being correct. Therefore for optimization reasons the checksum calculation might be disabled.

4.2.64 IP_UDP_EnableRxChecksum()

Description

Enables checksum verification of the checksum in the TCP header for incoming packets.

Prototype

```
void IP_TCP_EnableRxChecksum ( U8 IFace );
```

Parameter

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

Table 4.54: IP_TCP_EnableRxChecksum() parameter list

Additional information

In a typical network typically all data contained in a transferred frame has already been verified by the hardware checking the transmitted frames checksum and it is unlikely that data within this frame is corrupted if the frame checksum was verified as being correct. Therefore for optimization reasons the checksum calculation might be disabled.

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.

De-initialization should be done in the exact reversed order of initialization. This means terminating any created task that uses the IP API, terminating the IP_RxTask (if used), terminating the IP_Task and finally calling IP_DeInit() to close down the stack. The whole de-initialization should be done with Ethernet interrupts disabled and task switching disabled to prevent the de-initialization being interrupted by an Ethernet event.

De-init has to be supported by the driver as well. If your driver does not yet support IP_DeInit() you will end up in IP_Panic() . Please contact our support address and ask for IP_DeInit() support to be added to your driver.

Example

```
#include "IP.h"

void main(void) {
    IP_Init();
    //
    // Create IP tasks and use the stack
    //
    ...
    //
    // Disable Ethernet interrupt
    //
    OS_EnterRegion(); // Prevent task switching
    //
    // Terminate all application tasks that make use of the IP API
    //
    //
    // Terminate IP_RxTask first (if used) and IP_Task
    //
    IP_DeInit();
    OS_LeaveRegion(); // Allow task switching
}
```

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 25 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 than 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_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 4.55: IP_NI_ConfigPoll() parameter list

4.4.2 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 occurrence 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 <code>IP.h</code> and look for the "Driver capabilities" section.
CapsForcedValue	[IN] Value mask for the capabilities to force.

Table 4.56: IP_NI_ForceCaps() 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);
```

4.4.3 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 4.57: 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 PHY configuration functions

embOS/IP since version 3.02 has support for PHY drivers. Previous versions were using a fixed generic PHY driver. Existing `IP_X_Config()` configurations do not have to be modified. However when using `IP_PHY_*`() functions the new PHY driver concept using `IP_PHY_AddDriver()` on page 130 should be used and `IP_PHY_*`() functions should be called from the registered PHY configuration function.

4.5.1 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 4.58: 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 422.

4.5.2 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 4.59: 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 29 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 422.

4.5.3 IP_PHY_AddDriver()

Description

Adds a PHY driver and assigns it to an interface.

Prototype

```
void IP_PHY_AddDriver(      unsigned      IFaceId,
                           const IP_PHY_HW_DRIVER* pDriver,
                           const void*          pAccess,
                           IP_PHY_pfConfig      pfConfig );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
pDriver	Pointer to driver function table.
pAccess	Pointer to function table containing routines for hardware access, depending on the driver to add.
pfConfig	Callback to PHY config routine.

Table 4.60: IP_PHY_AddDriver() parameter list

Additional information

If a driver has already been added for the selected interface the driver will not be overwritten. The same applies for the hardware access functions and the config callback. This allows settings different parameters like the driver and access routines from different places.

Typically the network interface driver will try to add the generic PHY driver so it is not necessary to update an existing IP_X_Config() unless new IP_PHY_* functions shall be used or a driver other than the generic PHY driver shall be used.

Example

The following is an excerpt from an IP_Config_*.c file:

```

/*****
 *
 *      _ConfigPHY()
 *
 *  Function description
 *      Callback executed during the PHY init of the stack to configure
 *      PHY settings once the hardware interface has been initialized.
 *
 *  Parameters
 *      IFaceId: Zero-based interface index.
 */
static void _ConfigPHY(unsigned IFaceId) {
    //
    // Further PHY configuration can be added here by calling
    // IP_PHY_*() functions for generic or specific PHY configuration.
    //
}

/*****
 *
 *      IP_X_Config()
 *
 *  Function description
 *      This function is called by the IP stack during IP_Init().
 */
void IP_X_Config(void) {
    ...
    //
    // Add the generic PHY driver for interface #0 and register
    // a PHY config routine executed when the PHY driver is initialized.
    //
    IP_PHY_AddDriver(0, &IP_PHY_Driver_Generic, NULL, &_ConfigPHY);
    ...
}

```

4.5.4 IP_PHY_AddResetHook()

Description

Please refer to *IP_PHY_GENERIC_AddResetHook()* on page 413 for further information.

4.5.5 IP_PHY_ConfigAddr()

Description

Configures the PHY addr. to use.

Prototype

```
void IP_PHY_ConfigAddr( unsigned IFaceId,  
                        unsigned Addr );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
Addr	PHY addr.

Table 4.61: IP_PHY_ConfigAddr() parameter list

Additional information

New version of the old function *IP_NI_ConfigPHYAddr()* on page 128 that makes direct use of the PHY module.

4.5.6 IP_PHY_ConfigAltAddr()

Description

Sets a list of PHY addr. that can alternately be checked for the link state.

Prototype

```
void IP_PHY_ConfigAltAddr(          unsigned          IFaceId,
                                const IP_PHY_ALT_LINK_STATE_ADDR* pAltPhyAddr );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
pAltPhyAddr	List of alternate PHY addresses.

Table 4.62: IP_PHY_ConfigAltAddr() parameter list

Additional information

A typical setup would be using a switch where the first PHY/port uses the PHY addr. 0x01 and the second PHY/port uses the addr. 0x02. The PHY driver by default might only support one addr. to check the link state (e.g. on PHY addr. 0x01) and will ignore the link state on any other PHY addr. Using this alternate list of addr. these will be checked as well if supported by the driver.

Example

```
//
// PHY addresses of switch ports 2 - 4 (port 1 with addr. 0x01 will be
// found automatically).
//
const U8 aAltPhyAddr[] = { 0x02, 0x03, 0x04 };

const IP_PHY_ALT_LINK_STATE_ADDR AltPhyAddr = {
    aAltPhyAddr,
    SEGGER_COUNTOF(aAltPhyAddr)
};

void IP_X_Config(void) {
    ...
    IFaceId = IP_AddEtherInterface(DRIVER);
    IP_PHY_ConfigAltAddr(IFaceId, &AltPhyAddr);
    ...
}
```

4.5.7 IP_PHY_ConfigSupportedModes()

Description

Configures the duplex and speed modes that shall be supported. Typically the supported modes are the modes supported by the MAC.

Prototype

```
void IP_PHY_ConfigSupportedModes( unsigned IFaceId,
                                unsigned Modes );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
Modes	ORRed value of the following supported modes: <ul style="list-style-type: none"> - IP_PHY_MODE_10_HALF - IP_PHY_MODE_10_FULL - IP_PHY_MODE_100_HALF - IP_PHY_MODE_100_FULL - IP_PHY_MODE_1000_HALF - IP_PHY_MODE_1000_FULL

Table 4.63: IP_PHY_ConfigSupportedModes() parameter list

Additional information

New version of the old function *IP_SetSupportedDuplexModes()* on page 101 that makes direct use of the PHY module.

4.5.8 IP_PHY_ConfigUseStaticFilters()

Description

Tells the stack if using PHY static MAC filter is allowed.

Prototype

```
void IP_PHY_ConfigUseStaticFilters( unsigned IFaceId,  
                                   unsigned OnOff );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	0: Do not use the PHY static filters. 1: Use the PHY static filters.

Table 4.64: IP_PHY_ConfigUseStaticFilters() parameter list

Additional information

By default the stack is allowed to use PHY filters if available. Can be disabled using this function if a custom filtering by the user shall be used.

Needs to be used with a hardware interface (typically #0). Does have no effect when being used with virtual interfaces like Tail Tagging interfaces.

4.5.9 IP_PHY_DisableCheck()

Description

Disables PHY checks for all interfaces. This might be necessary for some PHYs that are not fully IEEE 802.3u compliant.

Prototype

```
void IP_PHY_DisableCheck( U32 Mask );
```

Parameter

Parameter	Description
Mask	ORRed bit mask of checks to disable: <ul style="list-style-type: none">- PHY_DISABLE_CHECK_ID- PHY_DISABLE_CHECK_LINK_STATE_AFTER_UP- PHY_DISABLE_WATCHDOG

Table 4.65: IP_PHY_DisableCheck() parameter list

4.5.10 IP_PHY_DisableCheckEx()

Description

Disables PHY checks for one interface. This might be necessary for some PHYs that are not fully IEEE 802.3u compliant.

Prototype

```
void IP_PHY_DisableCheckEx( unsigned IFaceId,
                           U32      Mask );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
Mask	ORRed bit mask of checks to disable: <ul style="list-style-type: none">- PHY_DISABLE_CHECK_ID- PHY_DISABLE_CHECK_LINK_STATE_AFTER_UP- PHY_DISABLE_WATCHDOG

Table 4.66: IP_PHY_DisableCheckEx() parameter list

4.5.11 IP_PHY_ReInit()

Description

Re-initializes the PHY.

Prototype

```
void IP_PHY_ReInit( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.67: IP_PHY_ReInit() parameter list

4.5.12 IP_PHY_SetWdTimeout()

Description

Configures the timeout of the watchdog that periodically checks if the PHY is in a good state.

Prototype

```
void IP_PHY_SetWdTimeout ( int ShiftCnt );
```

Parameter

Parameter	Description
<code>ShiftCnt</code>	Timeout comparison mask is $(1 \ll \text{ShiftCnt}) - 1$ in system ticks (typically 1 tick = 1ms). ($1 \ll \text{ShiftCnt}$) has to be larger than 5 (initial timeout) and can not be a larger ShiftCnt than 30.

Table 4.68: IP_PHY_SetWdTimeout() parameter list

Additional information

A PHY watchdog timeout might occur due to a link down of the interface if it had a link up before. In this case the stack resets the PHY as well to make sure it is not in a bad state and is kept functional.

4.6 Statistics functions

4.6.1 IP_STATS_EnableIFaceCounters()

Description

Enables statistic counters for a specific interface.

Prototype

```
void IP_STATS_EnableIFaceCounters ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.69: IP_STATS_EnableIFaceCounters() parameter list

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.2 IP_STATS_GetIFaceCounters()

Description

Retrieves a pointer to the statistic counters for a specific interface.

Prototype

```
IP_STATS_IFACE* IP_STATS_GetIFaceCounters ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.70: IP_STATS_GetIFaceCounters() parameter list

Return value

Success: Pointer to structure of type `IP_STATS_IFACE`.

Error : NULL

Additional information

`IP_SUPPORT_STATS_IFACE` or `IP_SUPPORT_STATS` needs to be enabled.

For further information please refer to *Structure IP_STATS_IFACE* on page 205.

4.6.3 IP_STATS_GetLastLinkStateChange()

Description

Retrieves the tick count when an interface entered its current state.

Prototype

```
U32 IP_STATS_GetLastLinkStateChange ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.71: IP_STATS_GetLastLinkStateChange() parameter list

Return value

Timestamp in system ticks (typically 1ms).

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.4 IP_STATS_GetRxBytesCnt()

Description

Retrieves the number of bytes received on an interface.

Prototype

```
U32 IP_STATS_GetRxBytesCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.72: IP_STATS_GetRxBytesCnt() parameter list

Return value

Number of bytes received on this interface.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.5 IP_STATS_GetRxDiscardCnt()

Description

Retrieves the number of packets received but discarded although they were O.K. .

Prototype

```
U32 IP_STATS_GetRxDiscardCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.73: IP_STATS_GetRxDiscardCnt() parameter list

Return value

Number of packets received but discarded although they were O.K. .

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.6 IP_STATS_GetRxErrCnt()

Description

Retrieves the number of receive errors.

Prototype

```
U32 IP_STATS_GetRxErrCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.74: IP_STATS_GetRxErrCnt() parameter list

Return value

Number of receive errors.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.7 IP_STATS_GetRxNotUnicastCnt()

Description

Retrieves the number of packets received on an interface that were not unicasts.

Prototype

```
U32 IP_STATS_GetRxNotUnicastCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.75: IP_STATS_GetRxNotUnicastCnt() parameter list

Return value

Number of packets received on this interface that were not unicasts.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.8 IP_STATS_GetRxUnicastCnt()

Description

Retrieves the number of unicast packets received on an interface.

Prototype

```
U32 IP_STATS_GetRxUnicastCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.76: IP_STATS_GetRxUnicastCnt() parameter list

Return value

Number of unicast packets received on this interface.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.9 IP_STATS_GetRxUnknownProtoCnt()

Description

Retrieves the number of unknown protocols received.

Prototype

```
U32 IP_STATS_GetRxUnknownProtoCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.77: IP_STATS_GetRxUnknownProtoCnt() parameter list

Return value

Number of unknown protocols received.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.10 IP_STATS_GetTxBytesCnt()

Description

Retrieves the number of bytes sent on an interface.

Prototype

```
U32 IP_STATS_GetTxBytesCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.78: IP_STATS_GetTxBytesCnt() parameter list

Return value

Number of bytes sent on this interface.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.11 IP_STATS_GetTxDiscardCnt()

Description

Retrieves the number of packets to send but discarded although they were O.K. .

Prototype

```
U32 IP_STATS_GetTxDiscardCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.79: IP_STATS_GetTxDiscardCnt() parameter list

Return value

Number of packets to send but discarded although they were O.K. .

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.12 IP_STATS_GetTxErrCnt()

Description

Retrieves the number of send errors on an interface.

Prototype

```
U32 IP_STATS_GetTxErrCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.80: IP_STATS_GetTxErrCnt() parameter list

Return value

Number of send errors.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.13 IP_STATS_GetTxNotUnicastCnt()

Description

Retrieves the number of packets sent on an interface that were not unicasts.

Prototype

```
U32 IP_STATS_GetTxNotUnicastCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.81: IP_STATS_GetTxNotUnicastCnt() parameter list

Return value

Number of packets sent on this interface that were not unicasts.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.6.14 IP_STATS_GetTxUnicastCnt()

Description

Retrieves the number of unicast packets sent on an interface.

Prototype

```
U32 IP_STATS_GetTxUnicastCnt ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.82: IP_STATS_GetTxUnicastCnt() parameter list

Return value

Number of unicast packets sent on this interface.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7 Other IP stack functions

4.7.1 IP_AddAfterInitHook()

Description

Adds a hook to a callback that is executed at the end of `IP_Init()` to allow adding initializations to be executed right after the stack itself has been initialized and all API can be used.

Prototype

```
void IP_AddAfterInitHook ( IP_HOOK_AFTER_INIT *pHook,
                          void (*pf)(void) );
```

Parameter

Parameter	Description
<code>pHook</code>	[IN] Pointer to static element of <code>IP_HOOK_AFTER_INIT</code> that can be internally used by the stack.
<code>pf</code>	[IN] Function pointer to the callback that will be executed.

Table 4.83: IP_AddAfterInitHook() parameter list

Additional information

Adding a callback to be executed right after `IP_Init()` can be helpful for various things. For example this allows using a centralized initialization that is not located in the main routine that calls `IP_Init()` and has to make use of IP API that is only valid to be used after `IP_Init()`.

Example

```
//
// Excerpt of content of IP_Config_*.c
//
static IP_HOOK_AFTER_INIT _Hook;

static void _Connect(void) {
    ...
}

void IP_X_Config(void) {
    ...
    IP_AddAfterInitHook(&_Hook, _Connect); // Register _Connect() to be
                                           // executed at end of IP_Init()
    ...
}

//
// Excerpt of content of main.c
//
void main(void) {
    ...
    IP_Init();
    ...
}
```

4.7.2 IP_AddEtherTypeHook()

Description

This function registers a callback to be called for received packets with the registered Ethernet type.

Prototype

```
void IP_AddEtherTypeHook(IP_HOOK_ON_ETH_TYPE* pHook,
                        int (*pf)(
                            unsigned IFaceId,
                            IP_PACKET* pPacket,
                            void* pBuffer,
                            U32 NumBytes ),
                        U16 Type);
```

Parameter

Parameter	Description
pHook	Hook resource of type IP_HOOK_ON_ETH_TYPE.
pf	Callback to call for the registered Ethernet type.
Type	Ethernet type that triggers the callback in host endianness.
IFaceId	Callback parameter: Zero-based interface index.
pPacket	Callback parameter: Pointer to packet that has been received.
pBuffer	Callback parameter: Pointer to start of packet data of received packet.
NumBytes	Callback parameter: NumBytes data in received packet.

Table 4.84: IP_AddEtherTypeHook() parameter list

Example

```
static IP_HOOK_ON_ETH_TYPE _Hook;

/*****
 *
 *      _OnARP()
 *
 *  Function description
 *      This function allocates a packet to mirror back a received ARP
 *      packet to the network. This is of no use but demonstrates how
 *      to use the API.
 *      The received packet will be handled regularly by the stack as
 *      well by returning IP_OK_TRY_OTHER_HANDLER.
 *
 *  Parameters
 *      IFaceId : Zero-based interface index.
 *      pPacket : Pointer to received packet.
 *      pBuffer : Pointer to start of data of the received packet.
 *      NumBytes: NumBytes data received in the packet.
 *
 *  Return value
 *      Original packet has not been changed and the stack shall
 *      process it: IP_OK_TRY_OTHER_HANDLER
 *      Original packet has been freed or reused by the callback:
 *      Other like IP_OK or IP_RX_ERROR.
 */
static int _OnARP(unsigned IFaceId, IP_PACKET* pPacket, void* pBuffer, U32 NumBytes) {
    IP_PACKET* pPacketOut;
    U8* p;

    pPacketOut = IP_AllocEtherPacket(IFaceId, NumBytes, &p);
    if (pPacketOut != NULL) {
        IP_MEMCPY(p, pBuffer, NumBytes);
        IP_SendEtherPacket(IFaceId, pPacketOut, NumBytes);
    }
    return IP_OK_TRY_OTHER_HANDLER;
}

/*****
 *
 *      MainTask()
 */
void MainTask(void) {
    IP_Init();
    IP_AddEtherTypeHook(&_Hook, _OnARP, 0x0806);
    ...
}
```

4.7.3 IP_AddOnPacketFreeHook()

Description

This function adds a hook function to the IP_HOOK_ON_PACKET_FREE list. Registered hooks will be called in case a packet gets freed.

Prototype

```
void IP_AddOnPacketFreeHook(IP_HOOK_ON_PACKET_FREE *pHook,  
                             void (*pf) (  
                                 IP_PACKET* pPacket ) );
```

Parameter

Parameter	Description
pHook	Element of type IP_HOOK_ON_PACKET_FREE to register.
pf	Callback that is notified on a packet free.
pPacket	Pointer to packet that has been freed (pointer is no longer valid).

Table 4.85: IP_AddOnPacketFreeHook() parameter list

4.7.4 IP_AddStateChangeHook()

Description

Adds a hook to a callback that is executed when the AdminState or HWState of an interface changes.

Prototype

```
void IP_AddStateChangeHook ( IP_HOOK_ON_STATE_CHANGE *pHook,
                             void (*pf)( unsigned IFaceId,
                                           U8      AdminState,
                                           U8      HWState ) );
```

Parameter

Parameter	Description
pHook	[IN] Pointer to static element of IP_HOOK_ON_STATE_CHANGE that can be internally used by the stack.
pf	[IN] Function pointer to the callback that will be executed.

Table 4.86: IP_AddStateChangeHook() parameter list

Additional information

A state change hook can be used to be notified about an interface disconnect that has not been triggered by the application. Typical example would be a peer that closes a dial-up connection and the application needs to get notified of this event to call a disconnect itself. Examples of this behavior can be found in the samples shipped with the stack.

Example

```
static IP_HOOK_ON_STATE_CHANGE _Hook;

static void _OnChange(unsigned IFaceId, U8 AdminState, U8 HWState) {
    ...
}

void main(void) {
    ...
    IP_AddStateChangeHook(&_Hook, _OnChange); // Register _OnState() to be
                                                // executed when interface changes.
    // Connect dial-up interface.
    ...
}
```

4.7.5 IP_Alloc()

Description

Thread safe memory allocation from main IP stack memory pool.

Prototype

```
void* IP_Alloc ( U32 NumBytesReq );
```

Parameter

Parameter	Description
NumBytesReq	Number of bytes to allocate.

Table 4.87: IP_Alloc() parameter list

Return value

Error: NULL

O.K. : Pointer to allocated memory

Additional information

Memory allocated with this function has to be freed with *IP_Free()* on page 169.

4.7.6 IP_AllocEtherPacket()

Description

Allocates a packet to store the raw data of an Ethernet packet of up to NumBytes at the location returned by ppBuffer.

Prototype

```
IP_PACKET* IP_AllocEtherPacket( unsigned IFaceId,
                                U32      NumBytes,
                                U8**     ppBuffer );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
NumBytes	Minimum buffer size the packet has to provide.
ppBuffer	Pointer where to store the pointer to the beginning of the packet buffer.

Table 4.88: IP_AllocEtherPacket() parameter list

Return value

O.K. : Pointer to packet allocated.

Error: NULL

4.7.7 IP_AllocEx()

Description

Thread safe memory allocation from a specific memory pool managed by the stack that has been added using *IP_AddMemory()* on page 57.

Prototype

```
void* IP_AllocEx ( U32* pBaseAddr,  
                  U32  NumBytesReq );
```

Parameter

Parameter	Description
pBaseAddr	Base address of the memory pool.
NumBytesReq	Number of bytes to allocate.

Table 4.89: IP_AllocEx() parameter list

Return value

Error: NULL

O.K. : Pointer to allocated memory

Additional information

Memory allocated with this function has to be freed with *IP_Free()* on page 169.

4.7.8 IP_Connect()

Description

Calls a previously registered callback that has been registered with *IP_SetIFaceConnectHook()* on page 197.

Prototype

```
int IP_Connect ( unsigned IFaceId );
```

Parameter

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

Table 4.90: IP_Connect() parameter list

Return value

0 : O.K. or no callback set.
Other: Error.

4.7.9 IP_Disconnect()

Description

Calls a previously registered callback that has been registered with *IP_SetIFaceDisconnectHook()* on page 198.

Prototype

```
int IP_Disconnect ( unsigned IFaceId );
```

Parameter

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

Table 4.91: IP_Disconnect() parameter list

Return value

0 : O.K. or no callback set.
Other: Error.

4.7.10 IP_Err2Str()

Description

Converts an error value to a printable string.

Prototype

```
const char * IP_Err2Str( int x );
```

Parameter

Parameter	Description
x	[IN] Error value other than 0.

Table 4.92: IP_Err2Str() parameter list

Return value

String describing the value.

4.7.11 IP_FindIFaceByIP()

Description

Tries to find out the interface number when only the IP address is known.

Prototype

```
int IP_FindIFaceByIP ( void*      pAddr,  
                      unsigned Len );
```

Parameter

Parameter	Description
<code>pAddr</code>	Pointer to a variable holding the address to find in host endianness.
<code>Len</code>	Length of address at pAddr.

Table 4.93: IP_FindIFaceByIP() parameter list

Return value

Interface not found: -1
Interface found : >= 0

Additional information

For the moment only IPv4 is supported.

Example

The following sample tries to find an interface that has previously been configured to a fixed IP address of 192.168.2.10 .

```
int IFaceId;  
U32 IPAddr;  
  
IPAddr = IP_BYTES2ADDR(192, 168, 2, 10);  
IFaceId = IP_FindIFaceByIP(&IPAddr, sizeof(IPAddr));
```


4.7.12 IP_Free()

Description

Thread safe memory free to IP stack memory pool.

Prototype

```
void IP_Free ( void *p );
```

Parameter

Parameter	Description
p	[IN] Pointer to memory block previously allocated with <i>IP_Alloc()</i> on page 162.

Table 4.94: IP_Free() parameter list

4.7.13 IP_FreePacket()

Description

Frees a packet back to the stack.

Prototype

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

Parameter

Parameter	Description
<code>pPacket</code>	Packet to free.

Table 4.95: IP_FreePacket() parameter list

Additional information

This routine can be used to typically free any allocated packet regardless of the API used to allocate it.

4.7.14 IP_GetAddrMask()

Description

Returns the IP address and the subnet mask of the device in host 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 in host order.
pMask	[OUT] Address to store the subnet mask in host order.

Table 4.96: IP_GetAddrMask() parameter list

4.7.15 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);  
}
```

4.7.16 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 4.97: 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);
}
```

4.7.17 IP_GetFreePacketCnt()

Description

Checks how many packets for a specific size or greater are currently available in the system.

Prototype

```
U32 IP_GetFreePacketCnt( U32 NumBytes );
```

Parameter

Parameter	Description
NumBytes	Minimum size of packets to find.

Table 4.98: IP_GetFreePacketCnt() parameter list

Return value

Number of packets available for this size.

4.7.18 IP_GetIFaceHeaderSize()

Description

Retrieves the size of the header necessary for the transport medium that is used by a specific interface.

Example: Ethernet: 14 bytes header + 2 bytes padding.

Prototype

```
U32 IP_GetIFaceHeaderSize( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.99: IP_GetIFaceHeaderSize() parameter list

Return value

Size of header for this interface.

4.7.19 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 4.100: IP_GetGWAddr() parameter list

Return value

The gateway address of the interface.

4.7.20 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 4.101: IP_GetHWAddr() parameter list

4.7.21 IP_GetIPAddr()

Description

Returns the IP address of the interface in host byte order (for example, 192.168.1.1 is returned as 0xC0A80101).

Prototype

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

Parameter

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

Table 4.102: IP_GetIPAddr() parameter list

Return value

The IP address of the interface in host byte order.

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);
}
```

4.7.22 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
<code>pPacket</code>	[IN] Pointer to an IP_PACKET structure.

Table 4.103: 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;
}

```

4.7.23 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 4.104: IP_GetRawPacketInfo() parameter list

Return value

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

4.7.24 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.

4.7.25 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 <code>IP_RX_HOOK</code> .

Table 4.105: 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++);
    }
}

```

4.7.26 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);  
}
```

4.7.27 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 );
```

Parameter

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

Table 4.106: IP_IFaceIsReadyEx() parameter list

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);  
}
```


4.7.28 IP_IsExpired()

Description

Checks if the given system timestamp has already expired.

Prototype

```
int IP_IsExpired ( I32 Time );
```

Parameter

Parameter	Description
Time	[IN] System timestamp as used by OS abstraction layer.

Table 4.107: IP_IsExpired() parameter list

Return value

1 Timestamp has expired.

0 Timestamp has not yet expired.

Example

```
U32 Timeout;

//
// Get current system time [ms] and timeout in one second.
//
Timeout = IP_OS_GET_TIME() + 1000;
//
// Wait until timeout expires.
//
do {
    OS_Delay(1);
} while (IP_IsExpired(Timeout) == 0);
```

4.7.29 IP_NI_GetAdminState()

Description

Retrieves the admin state of the given interface.

Prototype

```
int IP_NI_GetAdminState ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.108: IP_NI_GetAdminState() parameter list

Return value

0: Interface disabled

1: Interface enabled

-1: Invalid interface ID

4.7.30 IP_NI_GetIFaceType()

Description

Retrieves a short textual description of the interface type.

Prototype

```
int IP_NI_GetIFaceType ( unsigned IFaceId,
                        char*      pBuffer,
                        U32*      pNumBytes );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
pBuffer	Pointer to buffer where to store the string.
pNumBytes	Pointer to size of the buffer at pBuffer and where to store the length of the string (without termination).

Table 4.109: IP_NI_GetIFaceType() parameter list

Return value

O.K. : 0

Error: Other

Additional information

If the buffer is big enough this function will terminate the string in the buffer as well. The length of the string is always stored at [pNumBytes](#).

Example

```
char ac[10]; // Should be big enough to hold all short interface descriptions.
U32 NumBytes;

//
// Get the type of interface #0 .
//
NumBytes = sizeof(ac);
IP_NI_GetIFaceType(0, &ac[0], &NumBytes);
printf("Interface #0 is of type \"%s\"", &ac[0]);
```

4.7.31 IP_NI_GetState()

Description

Returns the hardware state of the interface.

Prototype

```
int IP_NI_GetState ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.110: IP_NI_GetState() parameter list

Return value

0: Interface is down

1: Interface is up

-1: Invalid interface ID

4.7.32 IP_NI_SetAdminState()

Description

Sets the AdminState of the interface.

Prototype

```
void IP_NI_SetAdminState ( unsigned IFaceId,  
                           int      AdminState );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
AdminState	Admin state to set.

Table 4.111: IP_NI_SetAdminState() parameter list

4.7.33 IP_NI_GetTxQueueLen()

Description

Retrieves the current length of the Tx queue of an interface.

Prototype

```
void IP_NI_GetTxQueueLen ( unsigned IFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.

Table 4.112: IP_NI_GetTxQueueLen() parameter list

Return value

Current Tx queue length: >= 0
Error: < 0

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7.34 IP_PrintIPAddr()

Description

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

Prototype

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

Parameter

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

Table 4.113: IP_PrintIPAddr() parameter list

Return value

Length of string stored into the buffer without string termination character.

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
}
```

4.7.35 IP_ResolveHost()

Description

Resolve a host name string to its IP addr. by using a configured DNS server.

Prototype

```
int IP_ResolveHost( const char *sHost,  
                   U32      *pIPAddr,  
                   U32      ms );
```

Parameter

Parameter	Description
<code>sHost</code>	[IN] Host name to resolve.
<code>pIPAddr</code>	[OUT] Pointer to where to store the resolved IP addr. in network order.
<code>ms</code>	[IN] Timeout in ms to wait for the DNS server to answer.

Table 4.114: IP_ResolveHost() parameter list

Return value

0 O.K., host name resolved.
< 0 Error: Could not resolve host name.

Additional information

In contrast to the standard socket function `gethostbyname()`, this function allows resolving a host name in a thread safe way and should therefore be used whenever possible.

The retrieved IP address will be returned in network order so it can be directly used with the BSD socket API.

4.7.36 IP_SendEtherPacket()

Description

Sends a previously allocated Ethernet packet.

Prototype

```
int IP_SendEtherPacket( unsigned   IFaceId,
                        IP_PACKET* pPacket,
                        U32         NumBytes );
```

Parameter

Parameter	Description
IFace	Zero-based interface index.
pPacket	Previously allocated Ethernet packet to send.
NumBytes	Number of bytes that have been stored in the packet buffer.

Table 4.115: IP_SendEtherPacket() parameter list

Return value

O.K. : 0

Error: Other

4.7.37 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
<code>IFace</code>	[IN] Zero-based interface index.
<code>pData</code>	[IN] Data packet that should be sent.
<code>Numbytes</code>	[IN] Length of data which should be sent.

Table 4.116: 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

4.7.38 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, <code>NULL</code> if do not care.
datalen	[IN] Length of data to attach to ping request.
pingseq	[IN] Ping sequence number.

Table 4.117: 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 795 for detailed information about logging.

4.7.39 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, <code>NULL</code> if do not care.
datalen	[IN] Length of data to attach to ping request.
pingseq	[IN] Ping sequence number.

Table 4.118: 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 795 for detailed information about logging.

4.7.40 IP_SetIFaceConnectHook()

Description

Sets a callback for an interface that is executed when *IP_Connect()* on page 165 is called.

Prototype

```
void IP_SetIFaceConnectHook ( unsigned IFaceId,
                             int (*pf) ( unsigned IFaceId ) );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
pf	[IN] Callback to set.

Table 4.119: IP_SetIFaceConnectHook() parameter list

Additional information

Typically for a pure Ethernet interface this functionality is not needed. Typically it is used with dial-up interfaces or interfaces that need more configurations to be set by the application to work.

4.7.41 IP_SetIFaceDisconnectHook()

Description

Sets a callback for an interface that is executed when *IP_Disconnect()* on page 166 is called.

Prototype

```
void IP_SetIFaceDisconnectHook ( unsigned IFaceId,  
                                int (*pf) ( unsigned IFaceId ) );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index.
pf	[IN] Callback to set.

Table 4.120: IP_SetIFaceDisconnectHook() parameter list

Additional information

Typically for a pure Ethernet interface this functionality is not needed. Typically it is used with dial-up interfaces or interfaces that need more configurations to be set by the application to work.

4.7.42 IP_SetPacketToS()

Description

Sets the ToS byte in the IP header for a packet to be sent via the zero-copy API.

Prototype

```
void IP_SetPacketToS ( IP_PACKET *pPacket,
                      U8      ToS );
```

Parameter

Parameter	Description
<code>pPacket</code>	[IN] Pointer to packet to send.
<code>ToS</code>	[IN] Value to set for the ToS byte in the IP header.

Table 4.121: IP_SetPacketToS() parameter list

4.7.43 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 <code>IP_RX_HOOK</code> .

Table 4.122: 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 182 for an example.

4.8 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.

The following data-structures are meant for public usage together with the documented API.

4.8.1 Structure IP_BSP_INSTALL_ISR_PARA

Description

Used to pass parameters for installing an ISR handler between driver and hardware specific callback.

Prototype

```
typedef struct {  
    void (*pfISR)(void);  
    int ISRIndex;  
    int Prio;  
} IP_BSP_INSTALL_ISR_PARA;
```

Member	Description
pfISR	Interrupt handler to register.
ISRIndex	Interrupt index given by the driver as reference. The index might differ from hardware to hardware.
Prio	Interrupt priority given by the driver as reference. Override this with a priority that best fits your system.

Table 4.123: Structure IP_BSP_INSTALL_ISR_PARA member list

4.8.2 Structure IP_BSP_API

Description

Used to set callbacks for a driver to call hardware specific functions that can not be handled in a generic way by the driver itself.

Prototype

```
typedef struct {
    void (*pfInit)          (unsigned IFaceId);
    void (*pfDeInit)        (unsigned IFaceId);
    void (*pfInstallISR) (unsigned IFaceId, IP_BSP_INSTALL_ISR_PARA* pPara);
} IP_BSP_API;
```

Member	Description
pfInit	Initialize port pins and clocks for Ethernet. Can be NULL.
pfDeInit	De-initialize port pins and clocks for Ethernet. Can be NULL.
pfInstallISR	Install driver interrupt handler. Can be NULL. For further information regarding IP_BSP_INSTALL_ISR_PARA please refer to <i>Structure IP_BSP_INSTALL_ISR_PARA</i> on page 202.

Table 4.124: Structure IP_BSP_API member list

Additional information

For further information about how this structure is used please refer to *IP_BSP_SetAPI()* on page 69.

4.8.3 Structure SEGGER_CACHE_CONFIG

Description

Used to pass cache configuration and callback function pointers to the stack.

Prototype

```
typedef struct {
    int    CacheLineSize;
    void (*pfDMB)      (void);
    void (*pfClean)     (void *p, unsigned NumBytes);
    void (*pfInvalidate)(void *p, unsigned NumBytes);
} SEGGER_CACHE_CONFIG;
```

Member	Description
<code>CacheLineSize</code>	Length of one cache line of the CPU. 0: No Cache. >0: Cache line size in bytes. Most Systems such as ARM9 use a 32 bytes cache line size.
<code>pfDMB</code>	Pointer to a callback function that executes a DMB (Data Memory Barrier) instruction to make sure all memory operations are completed. Can be NULL.
<code>pfClean</code>	Pointer to a callback function that executes a clean operation on cached memory. Can be NULL.
<code>pfInvalidate</code>	Pointer to a callback function that executes a clean operation on cached memory. Can be NULL.

Table 4.125: Structure SEGGER_CACHE_CONFIG member list

Additional information

For further information about how this structure is used please refer to *IP_CACHE_SetConfig()* on page 70.

4.8.4 Structure IP_STATS_IFACE

Description

Used to access the whole structure that can be accessed individually using the `IP_STATS_*` functions. Primary usage for these information is utilizing them for SNMP statistics, therefore their SNMP usage is explained.

Prototype

```
typedef struct {
    U32 LastLinkStateChange;
    U32 RxBytesCnt;
    U32 RxUnicastCnt;
    U32 RxNotUnicastCnt;
    U32 RxDiscardCnt;
    U32 RxErrCnt;
    U32 RxUnknownProtoCnt;
    U32 TxBytesCnt;
    U32 TxUnicastCnt;
    U32 TxNotUnicastCnt;
    U32 TxDiscardCnt;
    U32 TxErrCnt;
} IP_STATS_IFACE;
```

Member	Description (SNMP usage)
LastLinkStateChange	SNMP: ifLastChange [TimeTicks]. Needs to be converted into in 1/100 seconds since SNMP epoch.
RxBytesCnt	SNMP: ifInOctets [Counter].
RxUnicastCnt	SNMP: ifInUcastPkts [Counter].
RxNotUnicastCnt	SNMP: ifInNUcastPkts [Counter].
RxDiscardCnt	SNMP: ifInDiscards [Counter].
RxErrCnt	SNMP: ifInErrors [Counter].
RxUnknownProtoCnt	SNMP: ifInUnknownProtos [Counter].
TxBytesCnt	SNMP: ifOutOctets [Counter].
TxUnicastCnt	SNMP: ifOutUcastPkts [Counter].
TxNotUnicastCnt	SNMP: ifOutNUcastPkts [Counter].
TxDiscardCnt	SNMP: ifOutDiscards [Counter].
TxErrCnt	SNMP: ifOutErrors [Counter].

Table 4.126: Structure IP_STATS_IFACE member list

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. embOS/IP specific functions allow an easier or even extended usage of some socket operations. All API functions are described in this chapter.

5.1 API functions

The table below lists the available socket API functions.

Function	Description
Generic socket interface functions	
<code>accept()</code>	Accepts an incoming attempt on a socket.
<code>bind()</code>	Assigns a name to an unnamed socket.
<code>closesocket()</code>	Closes an existing socket.
<code>connect()</code>	Establishes a connection to a socket.
<code>gethostbyname()</code>	Resolves a host name into an IP address.
<code>getpeername()</code>	Returns the IP addressing information of the connected host.
<code>getsockname()</code>	Returns the current name for the specified socket.
<code>getsockopt()</code>	Returns the socket options.
<code>listen()</code>	Marks a socket as accepting connections.
<code>recv()</code>	Receives data from a connected socket.
<code>recvfrom()</code>	Receives a datagram and stores the source address.
<code>select()</code>	Checks if socket is ready.
<code>send()</code>	Sends data on a connected socket.
<code>sendto()</code>	Sends data to a specified address.
<code>setsockopt()</code>	Sets a socket option.
<code>shutdown()</code>	Disables sends or receives on a socket.
<code>socket()</code>	Creates an unbound socket.
embOS/IP specific socket interface functions	
<code>IP_SOCKET_GetAddrFam()</code>	Returns the IP version of a socket.
<code>IP_SOCKET_GetLocalPort()</code>	Returns the local port of a socket.
Helper macros	
<code>ntohl</code>	Converts a unsigned long value from network to host byte order.
<code>htonl</code>	Converts a unsigned long value from host byte order to network byte order.
<code>htons</code>	Converts a unsigned short value from host byte order to network byte order.
<code>ntohs</code>	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	[IN] Pointer to a variable with the max. length of socket address that can be stored. [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 234 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`.

Example

The following sample can be used to retrieve information about the accepted client:

```
struct sockaddr_in Client;

...
struct sockaddr_in Addr;
```

```
int AddrLen;

AddrLen = sizeof(Addr);
if ((hSock = accept(hSockListen, (struct sockaddr*)&Addr, &AddrLen)) ==
SOCKET_ERROR) {
    continue;    // Error
}
...
```

For example the peer IP address can then be retrieved in network endianness from `Addr.sin_addr.s_addr`.

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` or `SOCK_RAW`, 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.

To enable RAW socket support in the IP stack it is mandatory to call `IP_RAW_Add()` on page 93 during initialization of the stack.

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 231 for more information), 0 is returned.

You can use the `getsockopt()` function (refer to `getsockopt()` on page 219) to determine the status of the connect attempt.

The timeout for a connect attempt can be configured via the [Init](#) parameter of `IP_TCP_SetConnKeepaliveOpt()` on page 111.

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 (const char * name);
```

Parameter

Parameter	Description
name	[IN] 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 241 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);  
}  
}
```

Warning: gethostbyname() is not thread safe and should therefore only be used where absolutely necessary. If possible use the thread safe function IP_ResolveHost() instead.

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
<code>Socket</code>	[IN] A descriptor identifying a socket.
<code>pAddr</code>	[OUT] A pointer to a structure of type <code>sockaddr</code> in which the IP address information of the connected host should be stored.
<code>pAddrLen</code>	[IN] Pointer to a variable with the max. length of socket address that can be stored. [OUT] Pointer where to store the length of the socket address.

Table 5.7: getpeername() parameter list

Return value

0 on success.
-1 on failure.

Additional information

Refer to *Structure sockaddr* on page 238 for detailed information about the structure `sockaddr`.

Example

The following sample can be used to retrieve information about the peer host from an existing connection:

```
struct sockaddr_in Client;
int i;

...
if ((hSock = accept(hSockListen, &Addr, &AddrLen)) == SOCKET_ERROR) {
    continue;    // Error
}
i = sizeof(Client);
getpeername(hSock, (struct sockaddr*)&Client, &i);
...
```

For example the peer IP address can then be retrieved in network endianness from `Client.sin_addr.s_addr`.

5.1.7 getsockname()

Description

Returns the current address to which the socket is bound in the buffer pointed to by pAddr.

Prototype

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

Parameter

Parameter	Description
Socket	[IN] A descriptor identifying a socket.
pAddr	[OUT] A pointer to a structure of type <code>sockaddr</code> in which the IP address information of the connected host should be stored.
pAddrLen	[OUT] Max. size of address to return without exceeding the output buffer.

Table 5.8: getsockname() parameter list

Return value

0 on success.

-1 on failure.

Additional information

Refer to *Structure sockaddr* on page 238 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
<code>Socket</code>	[IN] A descriptor identifying a socket.
<code>Level</code>	[IN] Compatibility parameter for <code>setsockopt()</code> and <code>getsockopt()</code> . Use symbol <code>SOL_SOCKET</code> .
<code>Option</code>	[IN] The socket option which should be retrieved.
<code>pData</code>	[OUT] A pointer to the buffer in which the value of the requested option should be stored.
<code>DataLen</code>	[IN] The size of the data buffer.

Table 5.9: `getsockopt()` parameter list

Valid values for parameter Option

Value	Description
Standard option flags.	
<code>SO_ACCEPTCONN</code>	Indicates that socket is in listen mode.
<code>SO_DONTROUTE</code>	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.
<code>SO_KEEPAIVE</code>	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.
<code>SO_LINGER</code>	Indicates that linger on close is enabled.
<code>SO_NOSLOWSTART</code>	Indicates that suppress slow start on this socket is enabled.
<code>SO_TIMESTAMP</code>	Indicates that the TCP timestamp option is enabled.
embOS/IP socket options.	
<code>SO_ERROR</code>	Stores the latest socket error in <code>pData</code> and clears the error in socket structure.
<code>SO_MYADDR</code>	Stores the IP address of the used interface in <code>pData</code> .
<code>SO_RCVTIMEO</code>	Returns the timeout for <code>recv()</code> in ms. A return value of 0 indicates that no timeout is set.
<code>SO_NONBLOCK</code>	Gets sockets blocking status. Allows the caller to specify blocking or non-blocking IO that works the same as the other Boolean socket options. <code>pData</code> 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.
<code>IP_HDRINCL</code>	Checks if the IP header has to be included by the user for a RAW socket.

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
<code>IP_ERR_SEND_PENDING</code>	1	Packet to send is not sent yet.
<code>IP_ERR_MISC</code>	-1	Miscellaneous errors that do not have a specific error code.
<code>IP_ERR_TIMEDOUT</code>	-2	Operation timed out.
<code>IP_ERR_ISCONN</code>	-3	Socket is already connected.
<code>IP_ERR_OP_NOT_SUPP</code>	-4	Operation not supported for selected socket.
<code>IP_ERR_CONN_ABORTED</code>	-5	Connection was aborted.
<code>IP_ERR_WOULD_BLOCK</code>	-6	Socket is in non-blocking state and the current operation would block the socket if not in non-blocking state.
<code>IP_ERR_CONN_REFUSED</code>	-7	Connection refused by peer.
<code>IP_ERR_CONN_RESET</code>	-8	Connection has been reset.
<code>IP_ERR_NOT_CONN</code>	-9	Socket is not connected.
<code>IP_ERR_ALREADY</code>	-10	Socket already is in the requested state.
<code>IP_ERR_IN_VAL</code>	-11	Passed value for configuration is not valid.
<code>IP_ERR_MSG_SIZE</code>	-12	Message is too big to send.
<code>IP_ERR_PIPE</code>	-13	Socket is not in the correct state for this operation.
<code>IP_ERR_DEST_ADDR_REQ</code>	-14	Destination addr. has not been specified.
<code>IP_ERR_SHUTDOWN</code>	-15	Connection has been closed as soon as all data has been received upon a FIN request.
<code>IP_ERR_NO_PROTO_OPT</code>	-16	Unknown socket option for <code>setsockopt()</code> or <code>getsockopt()</code> .
<code>IP_ERR_NO_MEM</code>	-18	Not enough memory in the memory pool.
<code>IP_ERR_ADDR_NOT_AVAIL</code>	-19	No known path to send to the specified addr.
<code>IP_ERR_ADDR_IN_USE</code>	-20	Socket already has a connection to this addr. and port or is already bound to this addr.
<code>IP_ERR_IN_PROGRESS</code>	-22	Operation is still in progress.
<code>IP_ERR_NO_BUF</code>	-23	No internal buffer was available.
<code>IP_ERR_NOT SOCK</code>	-24	Socket has not been opened or has already been closed
<code>IP_ERR_FAULT</code>	-25	Generic error for a failed operation.
<code>IP_ERR_NET_UNREACH</code>	-26	No path to the desired network available.
<code>IP_ERR_PARAM</code>	-27	Invalid parameter to function.
<code>IP_ERR_LOGIC</code>	-28	Logical error that should not have happened.
<code>IP_ERR_NOMEM</code>	-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.
IP_ERR_QUEUE_FULL	-37	No more packets can be queued for sending. Typically caused by packets waiting for an ARP response to be fulfilled.

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 219 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 *socket()* on page 234 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 225.

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 231 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 234 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 231 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
<code>readfds</code>	See below.
<code>writefds</code>	
<code>exceptfds</code>	
<code>tv</code>	

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 milliseconds 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 231 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);
        IP_FD_SET(Socket, &readfds);           // Clear the set
                                                // 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 230 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: setsockopt() 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_KEEPAIVE	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. For keepalive behavior configuration please refer to <i>IP_TCP_SetConnKeepaliveOpt()</i> on page 111.
SO_LINGER	Controls the action taken when unsent messages are queued on a socket and a <code>closesocket()</code> is performed. Refer to <i>closesocket()</i> on page 212 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 <i>TCP zero-copy interface</i> on page 243 for detailed information.

Value	Description
SO_RCVTIMEO	Sets a timeout for <code>recv()</code> in ms like in Windows. This changes the behavior of <code>recv()</code> . <code>recv()</code> is by default a blocking function which only returns if data has been received. If a timeout is set <code>recv()</code> will return in case of data reception or timeout. By default, this socket option is disabled.
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. <code>pData</code> 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.
IP_HDRINCL	Configures if the IP header has to be included by the user or if the IP header is generated by the stack.

Return value

0 on success

Example

```
void _EnableKeepAlive(long sock) {  
    int v = 1;  
  
    setsockopt(sock, SOL_SOCKET, SO_KEEPAIVE, &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 except when Type is SOCK_RAW.

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
SOCK_RAW	RAW 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).

When receiving data from a socket of type `SOCK_STREAM` only up to the requested amount of data is consumed from the socket buffer upon calling a receive routine. Excess bytes of a message remain in the socket buffer and are available upon further calls to the receive routine.

When receiving data from a socket that is not of type `SOCK_STREAM` like a socket of type `SOCK_DGRAM` or `SOCK_RAW` one complete message (in the chunk as it was received) will be consumed and excess bytes of this message that are not read out of the buffer will be discarded and are not available for further calls to the receive routine.

`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.

`SOCK_RAW` sockets allow receiving data including network and IP header and allow sending of data either with or without specifying the IP header yourself. RAW sockets are operated the same way as `SOCK_DGRAM` sockets but allow the ability to receive data including the IP and protocol header and to implement your own protocol.

For using RAW sockets it is mandatory to call `IP_RAW_Add()` on page 93 during the initialization of the stack.

More information about RAW sockets can be found below.

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 219 and `setsockopt()` on page 231 for detailed information.

RAW sockets (receiving)

For RAW sockets the `Proto` parameter specifies the IP protocol that will be received using this socket. Protocols registered to be used with `IP_*_Add()` will be handled the stack and can not be used with RAW sockets at the same time. Using `IPPROTO_RAW` will receive data for any protocol not handled by the IP stack.

RAW sockets (sending)

For RAW sockets the `Proto` parameter specifies the IP protocol that will be entered into the IP header when sending data using this socket. Using `IPPROTO_RAW` for `Proto` for a sending socket results in the same as setting the socket option `IP_HDRINCL` for this socket by using `setsockopt()` on page 231 and requires the user to include his own IP header in the data to send.

5.1.18 IP_SOCKET_GetAddrFam()

Description

Returns the IP version of a socket (IPv4 or IPv6).

Prototype

```
U16 IP_SOCKET_GetAddrFam( int hSock );
```

Parameter

Parameter	Description
hSock	Socket handle.

Table 5.20: IP_SOCKET_GetAddrFam() parameter list

Return value

0: Invalid socket handle

AF_INET: IPv4 socket.

AF_INET6: IPv6 socket.

5.1.19 IP_SOCKET_GetLocalPort()

Description

Returns the local port of a socket.

Prototype

```
U16 IP_SOCKET_GetLocalPort( int hSock );
```

Parameter

Parameter	Description
hSock	Socket handle.

Table 5.21: IP_SOCKET_GetLocalPort() parameter list

Return value

>0: OK. Local port number of the socket in network byte order.

0: Error. Socket not available or no local port bound to socket.

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
<code>sa_family</code>	Address family. Normally <code>AF_INET</code> .
<code>sa_data</code>	The character array <code>sa_data</code> contains the destination address and port number for the socket.

Table 5.22: 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 239 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
<code>sin_family</code>	Address family. Normally AF_INET.
<code>sin_port</code>	Port number for the socket.
<code>sin_addr</code>	Structure of type <code>in_addr</code> . The structure represents a 4-byte number that represents one digit in an IP address per byte.
<code>sin_zero</code>	<code>sin_zero</code> member is unused.

Table 5.23: Structure sockaddr_in member list

Example

Refer to `connect()` on page 213 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 5.24: 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 5.25: 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 5.26: 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. Therefore, `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
<code>IP_OK</code>	0	Data handled, packet can be freed.
<code>IP_OK_KEEP_PACKET</code>	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 231.

6.4 API functions

Function	Description
IP_TCP_Alloc()	Allocates a packet buffer.
IP_TCP_AllocEx()	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 `NumBytes` bytes of TCP data, plus TCP, IP and MAC headers.

Prototype

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

Parameter

Parameter	Description
<code>NumBytes</code>	[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 1516 bytes). Refer to `IP_AddBuffers()` on page 53 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_AllocEx()

Description

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

Prototype

```
IP_PACKET * IP_TCP_Alloc (int NumBytes, int NumBytesHeader);
```

Parameter

Parameter	Description
<code>NumBytes</code>	[IN] Length of the data which should be sent.
<code>NumBytesHeader</code>	[IN] Size of all headers (Ethernet + IPvX + TCPvX).

Table 6.4: IP_TCP_AllocEx() parameter list

Return value

Success: Returns a pointer to the allocated buffer.

Error: NULL

Additional information

For further information please refer to *IP_TCP_Alloc()* on page 248.

6.4.3 IP_TCP_Free()

Description

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

Prototype

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

Parameter

Parameter	Description
<code>pPacket</code>	[IN] Pointer to the <code>IP_Packet</code> structure.

Table 6.5: IP_TCP_Free() parameter list

6.4.4 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 6.6: 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. Therefore, `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.

6.4.5 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 <code>IP_Packet</code> structure.

Table 6.7: 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 UDP/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 45 for detailed information 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. Therefore, `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, htonl(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
<code>IP_OK</code>	0	Data handled. embOS/IP will free the packet.
<code>IP_OK_KEEP_PACKET</code>	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 269.

7.4 API functions

Function	Description
<code>IP_UDP_Alloc()</code>	Returns a pointer to a packet buffer big enough for the specified sizes.
<code>IP_UDP_Close()</code>	Closes a UDP connection handle.
<code>IP_UDP_FindFreePort()</code>	Returns a free local port number.
<code>IP_UDP_Free()</code>	Frees the buffer which was used for a packet.
<code>IP_UDP_GetDataSize()</code>	Returns size of data contained in the received UDP packet.
<code>IP_UDP_GetDataPtr()</code>	Returns pointer to data contained in the received UDP packet.
<code>IP_UDP_GetDestAddr()</code>	Retrieves the IP address of the destination of the given UDP packet.
<code>IP_UDP_GetFPort()</code>	Extracts foreign port information from a UDP packet.
<code>IP_UDP_GetIFIndex()</code>	Extract the interface on which the packet has been received.
<code>IP_UDP_GetLPort()</code>	Extracts local port information from a UDP packet.
<code>IP_UDP_GetSrcAddr()</code>	Retrieves the IP address of the sender of the given UDP packet.
<code>IP_UDP_Open()</code>	Creates a UDP connection handle.
<code>IP_UDP_OpenEx()</code>	Creates a UDP connection handle.
<code>IP_UDP_Send()</code>	Sends an UDP packet to a specified host.
<code>IP_UDP_SendAndFree()</code>	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. Therefore, `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 endianness.

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 269 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_GetDataSize()

Description

Returns size of data contained in the received UDP packet.

Prototype

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

Parameter

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

Table 7.6: IP_UDP_GetDataSize() parameter list

Return value

Size of data contained in received UDP packet.

7.4.6 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 7.7: IP_UDP_GetDataPtr() parameter list

Return value

Pointer to the data part of the UDP packet.

7.4.7 IP_UDP_GetDestAddr()

Description

Extracts destination address information from a UDP packet.

Prototype

```
void IP_UDP_GetDestAddr( const IP_PACKET * pPacket,  
                        void          * pDestAddr,  
                        int           AddrLen );
```

Parameter

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

Table 7.8: IP_UDP_GetDestAddr() parameter list

7.4.8 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 7.9: IP_UDP_GetFPort() parameter list

Return value

Foreign port extracted from the packet.

7.4.9 IP_UDP_GetIFIndex()

Description

Extracts the interface information from a UDP packet.

Prototype

```
unsigned IP_UDP_GetIFIndex ( const IP_PACKET * pPacket );
```

Parameter

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

Table 7.10: IP_UDP_GetIFIndex() parameter list

Return value

Zero-based interface index on which the packet was received.

7.4.10 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 7.11: IP_UDP_GetLPort() parameter list

Return value

Local port extracted from the packet.

7.4.11 IP_UDP_GetSrcAddr()

Description

Extracts source address information from a 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 7.12: IP_UDP_GetSrcAddr() parameter list

7.4.12 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      FAddr,
                        U16          fport,
                        U16          lport,
                        int(*routine) (IP_PACKET *, void * pContext),
                        void *       pContext );
```

Parameter

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

Table 7.13: IP_UDP_Open() parameter list

Return value

Success: Returns a pointer to the UDP connection handle.

Error: NULL

Additional information

The parameters [FAddr](#), [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 save the return value.

7.4.13 IP_UDP_OpenEx()

Description

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

Prototype

```
IP_UDP_CONN IP_UDP_OpenEx( IP_ADDR      FAddr,
                           U16          fport,
                           IP_ADDR      LAddr,
                           U16          lport,
                           int(*routine) (IP_PACKET *, void * pContext),
                           void *       pContext );
```

Parameter

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

Table 7.14: IP_UDP_OpenEx() parameter list

Return value

Success: Returns a pointer to the UDP connection handle.

Error: NULL

Additional information

The parameters [FAddr](#), [fport](#), [LAddr](#) 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_OpenEx\(\)](#). 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 save the return value.

7.4.14 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 endianness.
fport	[IN] Foreign port in host endianness.
lport	[IN] Local port in host endianness.
pPacket	[IN] Data which should be sent to the target host.

Table 7.15: IP_UDP_Send() parameter list

Return value

On success: 0 (1 if packet is queued for sending)

On error: <0 error code

Additional information

The packet [pPacket](#) has to be allocated by calling `IP_UDP_Alloc()`. Refer to *IP_UDP_Alloc()* on page 258 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 269 for more information about creating an UDP connection.

`IP_UDP_Send()` does not free the packet in case of an error. In this case it is the responsibility of the application programmer to either free the packet using `IP_UDP_Free()` or to try sending the packet again.

7.4.15 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 endianness.
fport	[IN] Foreign port in host endianness.
lport	[IN] Local port in host endianness.
pPacket	[IN] Data which should be sent to the target host.

Table 7.16: IP_UDP_SendAndFree() parameter list

Return value

On success: 0 (1 if packet is queued for sending)
On error: <0 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 269 for more information about creating an UDP connection.

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

Chapter 8

RAW zero-copy interface

Transferring RAW data can be used via socket functions or the zero-copy interface which is described in this chapter.

8.1 RAW zero-copy

The RAW 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.

RAW zero-copy API functions are intended to assist the development of higher-performance embedded network applications by allowing the application direct access to the 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.

To enable RAW socket support in the IP stack it is mandatory to call `IP_RAW_Add()` on page 93 during initialization of the stack.

8.1.1 Allocating, freeing and sending packet buffers

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

`IP_RAW_Alloc()` allocates a packet buffer from the pool of packet buffers on the stack and `IP_RAW_Free()` frees a packet buffer. Applications using the RAW 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_RAW_Send()` and `IP_RAW_SendAndFree()`, send a packet buffer of data using a specific protocol or sending pure data which requires the user to include his own IP header. The RAW 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. Therefore, `IP_RAW_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_RAW_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_RAW_Free()` should be called to free the packet.

8.1.2 Callback function

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

8.2 Sending data with the RAW zero-copy API

To send data with the RAW 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.

8.2.1 Allocating a packet buffer

The first step in using the RAW zero-copy API to send data is to allocate a packet buffer from the stack using the `IP_RAW_Alloc()` function. This function takes the maximum length of the data you intend to send in the buffer and if the IP header will be written by the stack or by yourself as arguments 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_RAW_Alloc(0, DataLen, 0); // Stack will write IP header
if (pPacket == NULL) {
    // Error, could not allocate packet buffer
}
```

This limits how much data you can send in one call using the RAW zero-copy API, as the data sent in one call to `IP_RAW_Send()` must fit in a single packet buffer. The actual limit is determined by the big packet buffer size, less typically 34 bytes for protocol headers (14 bytes for Ethernet header, 20 bytes IP header). If you try to request a larger buffer than this, `IP_RAW_Alloc()` returns `NULL` to indicate that it cannot allocate a sufficiently large buffer.

If you decide to provide the IP header yourself you can allocate a packet buffer the following way:

```
pPacket = IP_RAW_Alloc(0, DataLen, 1);
```

In this case the packet size allocate limit is determined by the big packet buffer size, less typically 14 bytes for the Ethernet header.

8.2.2 Filling the allocated buffer with data

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

Depending on if you decided to provide your own IP header you will have to store this data starting at `pPacket->pData` as well.

8.2.3 Sending the packet

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

```
#define PROTOCOL 1 // ICMP
#define DEST_ADDR 0xC0A80101

e = IP_RAW_Send(0, DEST_ADDR, PROTOCOL, pPacket);
if (e < 0) {
    IP_RAW_Free(pPacket);
}
```

This function sends the packet specifying the ICMP protocol in the IP header, 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_RAW_SendAndFree()` if the packet should be freed automatically in any case.

In case you intend to provide your own IP header the protocol passed has to be `IPPROTO_RAW`. This prevents the stack to generate and include a header on its own.

8.3 Receiving data with the RAW zero-copy API

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

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

8.3.1 Writing a callback function

Using the RAW 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 protocol. 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 (including network and IP header), 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
<code>IP_OK</code>	0	Data handled. embOS/IP will free the packet.
<code>IP_OK_KEEP_PACKET</code>	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 8.1: embOS/IP RAW 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_RAW_Free()`.

8.3.2 Registering the callback function

The application must also inform the stack of the callback function. This is done by calling the `IP_RAW_Open()` function. The following code fragment illustrates the use of this option to register a callback function named `RxUpCall()` for the ICMP protocol:

```
#define PROTOCOL 1 // ICMP
```

```
IP_RAW_Open(0L /* any foreign host */, 0L /* any local host */, PROTOCOL, RxUpCall, 0L /* any tag */);
```

The function `IP_RAW_Open()` is described in *IP_RAW_Open()* on page 287 .

To receive ICMP packets the ICMP protocol has not to be added to the stack by calling `IP_ICMP_Add()`. Protocols known to the stack and added for handling through the stack can not be used with the RAW zero-copy API.

8.4 API functions

Function	Description
<code>IP_RAW_Alloc()</code>	Returns a pointer to a packet buffer big enough for the specified sizes.
<code>IP_RAW_Close()</code>	Closes a RAW connection handle.
<code>IP_RAW_Free()</code>	Frees the buffer which was used for a packet.
<code>IP_RAW_GetDataPtr()</code>	Returns pointer to data contained in the received RAW packet.
<code>IP_RAW_GetDataSize()</code>	Retrieves the payload size in the packet.
<code>IP_RAW_GetDestAddr()</code>	Retrieves the IP address of the destination of the given RAW packet.
<code>IP_RAW_GetIFIndex()</code>	Extract the interface on which the packet has been received.
<code>IP_RAW_GetSrcAddr()</code>	Retrieves the IP address of the sender of the given RAW packet.
<code>IP_RAW_Open()</code>	Creates a RAW connection handle.
<code>IP_RAW_Send()</code>	Sends a RAW packet to a specified host.
<code>IP_RAW_SendAndFree()</code>	Sends a RAW packet to a specified host and frees the packet.

Table 8.2: embOS/IP RAW zero-copy API function overview

8.4.1 IP_RAW_Alloc()

Description

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

Prototype

```
IP_PACKET * IP_RAW_Alloc( unsigned IFaceId,
                          int      NumBytes,
                          int      IpHdrIncl );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available interfaces.
NumBytes	[IN] Length of the data which should be sent.
IpHdrIncl	[IN] Specifies if the IP header is generated or has to be provided by the user. 0: Header generated by the stack; 1: Header to be provided in the packet data by the user.

Table 8.3: IP_RAW_Alloc() parameter list

Return value

Success: Returns a pointer to the allocated buffer.

Error: NULL

Additional information

Applications using the RAW 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 RAW 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. Therefore, `IP_RAW_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_RAW_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_RAW_Free()` should be called to free the packet.

8.4.2 IP_RAW_Close()

Description

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

Prototype

```
void IP_RAW_Close( IP_RAW_CONN Con );
```

Parameter

Parameter	Description
Con	[IN] RAW connection handle.

Table 8.4: IP_RAW_Close() parameter list

8.4.3 IP_RAW_Free()

Description

Frees the buffer which was used for a packet.

Prototype

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

Parameter

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

Table 8.5: IP_RAW_Free() parameter list

8.4.4 IP_RAW_GetDataPtr()

Description

Returns pointer to data contained in the received RAW packet.

Prototype

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

Parameter

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

Table 8.6: IP_RAW_GetDataPtr() parameter list

Return value

Pointer to the data part of the packet.

Additional information

The data pointer returned points to the start of the network header. Therefore typically 34 bytes header (14 bytes Ethernet header, 20 bytes IP header) are included.

8.4.5 IP_RAW_GetDataSize()

Description

Returns size of the payload in the received RAW packet.

Prototype

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

Parameter

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

Table 8.7: IP_RAW_GetDataSize() parameter list

Return value

Number of data bytes received in the packet.

8.4.6 IP_RAW_GetDestAddr()

Description

Extracts destination address information from a RAW packet.

Prototype

```
void IP_RAW_GetDestAddr( const IP_PACKET * pPacket,  
                        void          * pDestAddr,  
                        int           AddrLen );
```

Parameter

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

Table 8.8: IP_RAW_GetDestAddr() parameter list

8.4.7 IP_RAW_GetIFIndex()

Description

Extracts the interface information from a RAW packet.

Prototype

```
unsigned IP_RAW_GetIFIndex ( const IP_PACKET * pPacket );
```

Parameter

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

Table 8.9: IP_RAW_GetIFIndex() parameter list

Return value

Zero-based interface index on which the packet was received.

8.4.8 IP_RAW_GetSrcAddr()

Description

Extracts source address information from a RAW packet.

Prototype

```
void IP_RAW_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 8.10: IP_RAW_GetSrcAddr() parameter list

8.4.9 IP_RAW_Open()

Description

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

Prototype

```
IP_RAW_CONN IP_RAW_Open( IP_ADDR      FAddr,
                          IP_ADDR      LAddr,
                          U8           Protocol,
                          int(*routine) (IP_PACKET *, void * pContext),
                          void *       pContext );
```

Parameter

Parameter	Description
FAddr	[IN] Foreign IP address.
LAddr	[IN] Local IP address.
Protocol	[IN] IP protocol.
(*routine)	[IN] Callback function which is called when a packet of protocol Protocol is received.
pContext	[IN/OUT] Application defined context pointer.

Table 8.11: IP_RAW_Open() parameter list

Return value

Success: Returns a pointer to the RAW connection handle.

Error: NULL

Additional information

The parameters [FAddr](#) and [LAddr](#) can be set to 0 as a wild card, which enables the reception of broadcast packets. To enable reception of any protocol use `IPPROTO_RAW` for [Protocol](#). The callback handler function is called with a pointer to a received protocol and a copy of the data pointer which is passed to `IP_RAW_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 packet.

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

8.4.10 IP_RAW_Send()

Description

Send a RAW packet to a specified host.

Prototype

```
int IP_RAW_Send( int          IFace,
                 IP_ADDR      FHost,
                 U8           Protocol,
                 IP_PACKET * pPacket );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based index of available interfaces.
FHost	[IN] IP address of the target host in network endianness.
Protocol	[IN] Protocol that will be used in the IP header generated by the stack.
pPacket	[IN] Packet that should be sent to the target host.

Table 8.12: IP_RAW_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_RAW_Alloc()`. Refer to *IP_RAW_Alloc()* on page 279 for detailed information.

If you expect to get any response to this packet you should have opened a RAW connection prior to calling `IP_RAW_Send()`. Refer to *IP_RAW_Open()* on page 287 for more information about creating a RAW connection.

`IP_RAW_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_RAW_Free()` should be called to free the packet.

8.4.11 IP_RAW_SendAndFree()

Description

Send a RAW packet to a specified host and frees the packet.

Prototype

```
int IP_RAW_SendAndFree( int      IFace,
                       IP_ADDR  FHost,
                       U8       Protocol,
                       IP_PACKET * pPacket );
```

Parameter

Parameter	Description
IFace	[IN] Zero-based index of available interfaces.
FHost	[IN] IP address of the target host in network endianness.
Protocol	[IN] Protocol that will be used in the IP header generated by the stack.
pPacket	[IN] Packet that should be sent to the target host.

Table 8.13: IP_RAW_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_RAW_Alloc()`. Refer to `IP_RAW_Alloc()` on page 279 for detailed information.

If you expect to get any response to this packet you should have opened a RAW connection prior to calling `IP_RAW_Send()`. Refer to `IP_RAW_Open()` on page 287 for more information about creating a RAW connection.

Packets are freed by calling `IP_RAW_SendAndFree()`. Therefore, no call of `IP_RAW_Free()` is required.

Chapter 9

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.

9.1 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.

9.2 API functions

Function	Description
<code>IP_DHCPC_Activate()</code>	Activates the DHCP client.
<code>IP_DHCPC_ConfigAlwaysStartInit()</code>	Configure to always start with DISCOVER.
<code>IP_DHCPC_ConfigOnActivate()</code>	Configure behavior on activate.
<code>IP_DHCPC_ConfigOnFail()</code>	Configure behavior on communication error.
<code>IP_DHCPC_ConfigOnLinkDown()</code>	Configure behavior on interface link down.
<code>IP_DHCPC_GetState()</code>	Returns the state of the DHCP client.
<code>IP_DHCPC_Halt()</code>	Stops all DHCP client activity.
<code>IP_DHCPC_Renew()</code>	Check configuration with server.
<code>IP_DHCPC_SetCallback()</code>	Sets a callback for an interface.
<code>IP_DHCPC_SetClientId()</code>	Sets the client ID field.

Table 9.1: embOS/IP DHCP client interface function overview

9.2.1 IP_DHCP_Activate()

Description

Activates the DHCP client.

Prototype

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

Parameter

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

Table 9.2: IP_DHCP_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_DHCP_Activate(0, "Target", NULL, NULL);
// Illegal function call
char ac;
sprintf(ac, "Target%d", Index);
IP_DHCP_Activate(0, ac, NULL, NULL);
// Correct function call
static char ac;
sprintf(ac, "Target%d", Index);
IP_DHCP_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.
```

9.2.2 IP_DHCPConfigAlwaysStartInit()

Description

Configures if the client always starts with INIT phase, sending a DISCOVER packet, even if an IP was configured for the interface before.

Prototype

```
void IP_DHCPConfigAlwaysStartInit (int IFaceId,
                                   U8  OnOff );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index to configure.
OnOff	[IN] 0: Off; 1: On.

Table 9.3: IP_DHCPConfigAlwaysStartInit() parameter list

9.2.3 IP_DHCPConfigOnActivate()

Description

Configures behavior regarding currently set parameters of an interface when the DHCP client is activated on this interface.

Prototype

```
void IP_DHCPConfigOnActivate( int IFaceId,
                             U8  Mode );
```

Parameter

Parameter	Description
<code>IFaceId</code>	[IN] Zero-based interface index to configure.
<code>Mode</code>	[IN] Mode to configure. The modes that can be setup are listed below.

Table 9.4: IP_DHCPConfigOnActivate() parameter list

Modes

Mode	Description
<code>DHCP_RESET_CONFIG</code>	Reset interface when activating the DHCP client on this interface to avoid using old settings longer than necessary. Default.
<code>DHCP_USE_STATIC_CONFIG</code>	Keep previous static configuration, if any, as fallback configuration as long as no new configuration has been received from a server.

Table 9.5: IP_DHCPConfigOnActivate() mode list

Additional information

This function needs to be called before activating the DHCP client for an interface using *IP_DHCPActivate()* on page 294. Please be aware that activating the DHCP client with a static configured IP address instructs the DHCP client to try to request this address from the server. In case *IP_DHCPConfigOnFail()* on page 297 is configured to use `DHCP_RESET_CONFIG` (default) it might happen that the static IP will be reset if no server is reachable or the IP addr. is declined from a server.

9.2.4 IP_DHCPConfigOnFail()

Description

Configures behavior regarding currently set parameters of an interface when the DHCP client fails in communication to negotiate a configuration with a server.

Prototype

```
void IP_DHCPConfigOnFail( int IFaceId,
                          U8  Mode );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based interface index to configure.
Mode	[IN] Mode to configure. The modes that can be setup are listed below.

Table 9.6: IP_DHCPConfigOnFail() parameter list

Modes

Mode	Description
DHCP_RESET_CONFIG	Reset interface to avoid using old settings longer than necessary as they might interfere with other DHCP clients in this network. Default.
DHCP_USE_STATIC_CONFIG	Setup previous static configuration, if any, as fallback configuration to remain accessible.
DHCP_USE_DHCP_CONFIG	Keep previously received DHCP configuration. Not recommended as it might interfere with other DHCP clients in this network.

Table 9.7: IP_DHCPConfigOnFail() mode list

Additional information

This function shall be called before activating the DHCP client for an interface using *IP_DHCPActivate()* on page 294.

9.2.5 IP_DHCPConfigOnLinkDown()

Description

Configures behavior regarding currently set parameters of an interface when the DHCP client is activated on this interface and the link goes down.

Prototype

```
void IP_DHCPConfigOnLinkDown( int IFaceId,
                              U32 Timeout,
                              U8  Mode );
```

Parameter

Parameter	Description
<code>IFaceId</code>	[IN] Zero-based interface index to configure.
<code>Timeout</code>	[IN] Timeout to wait before reacting on link down [ms].
<code>Mode</code>	[IN] Mode to configure. The modes that can be setup are listed below.

Table 9.8: IP_DHCPConfigOnLinkDown() parameter list

Modes

Mode	Description
<code>DHCP_RESET_CONFIG</code>	Reset interface when link goes down on this interface to avoid using old settings longer than necessary as the target might be connected to another network. Default.
<code>DHCP_USE_STATIC_CONFIG</code>	Setup previous static configuration, if any, as fallback configuration on link down to allow a quick start once the link goes up again.
<code>DHCP_USE_DHCP_CONFIG</code>	Keep previously received DHCP configuration on link down as long as the configuration is not declined or a new configuration is received once link on this interface is up again.

Table 9.9: IP_DHCPConfigOnLinkDown() mode list

Additional information

This function shall be called before activating the DHCP client for an interface using *IP_DHCPActivate()* on page 294.

9.2.6 IP_DHCP_GetState()

Description

Returns the state of the DHCP client.

Prototype

```
int IP_DHCP_GetState( int IFIndex );
```

Parameter

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

Table 9.10: IP_DHCP_GetState() parameter list

Return value

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

9.2.7 IP_DHCPC_Halt()

Description

Stops all DHCP activity on a network interface.

Prototype

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

Parameter

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

Table 9.11: IP_DHCPC_Halt() parameter list

9.2.8 IP_DHCP_Renew()

Description

Sends a request with the currently in use DHCP configuration to the DHCP server to check if the configuration is still valid.

Prototype

```
void IP_DHCP_Renew( int IFaceId );
```

Parameter

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

Table 9.12: IP_DHCP_Renew() parameter list

9.2.9 IP_DHCP_SetCallback()

Description

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

Prototype

```
void IP_DHCP_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 9.13: IP_DHCP_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.

9.2.10 IP_DHCP_SetClientId()

Description

This function allows the caller to set the client ID field content for the client.

Prototype

```
void IP_DHCP_SetClientId(          int      IFaceId,
                                const U8    *pClientId,
                                unsigned ClientIdLen );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index number of available network interfaces.
pClientId	[IN] Pointer to client ID to use. Will not be copied.
ClientIdLen	[IN] Length of client ID.

Table 9.14: IP_DHCP_SetClientId() parameter list

Additional information

Typically a DHCP server will recognize a client based on its MAC address. A client ID can be included by the client when communicating with the server for identification if needed. Please be aware that one byte is prepend that contains the type of the ID. The client ID will not be copied into the stack, therefore you need to make sure that the memory will be available even after the call. A good and a bad example is shown below.

Bad example

```
U8 ClientID[7];          // 1 byte type + 6 bytes MAC address.

ClientID[0] = 0x01;      // Type = Ethernet.
IP_GetHWAddr(0, &ClientID[1], sizeof(ClientID) - 1);
IP_DHCP_SetClientId(0, ClientID, sizeof(ClientID));
```

Good example

```
static U8 ClientID[7];   // 1 byte type + 6 bytes MAC address.

ClientID[0] = 0x01;      // Type = Ethernet.
IP_GetHWAddr(0, &ClientID[1], sizeof(ClientID) - 1);
IP_DHCP_SetClientId(0, ClientID, sizeof(ClientID));
```


Chapter 10

DHCP server (Add-on)

The embOS/IPThis implementation of the DHCP server is an optional extension to embOS/IP. It allows setting up a Dynamic Host Control Protocol (DHCP) server that seamlessly integrates with embOS/IP. All API functions are described in this chapter.

10.1 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.

Further information can be found in the chapter *DHCP backgrounds* on page 292 in the description of the DHCP client.

10.2 API functions

Function	Description
<code>IP_DHCPS_ConfigDNSAddr()</code>	Configure the DNS servers to distribute.
<code>IP_DHCPS_ConfigGWAddr()</code>	Configure the gateway to distribute.
<code>IP_DHCPS_ConfigMaxLeaseTime()</code>	Configure the max. lease time to grant.
<code>IP_DHCPS_ConfigPool()</code>	Configures the IP pool to use.
<code>IP_DHCPS_Halt()</code>	Halts the DHCP server.
<code>IP_DHCPS_Init()</code>	Initializes the DHCP server.
<code>IP_DHCPS_Start()</code>	Starts the DHCP server.

Table 10.1: embOS/IP DHCP server interface function overview

10.2.1 IP_DHCPConfigDNSAddr()

Description

Configures DNS servers to assign to clients.

Prototype

```
int IP_DHCPConfigDNSAddr ( unsigned IFIndex,
                           U32      *paDNSAddr,
                           U8        NumServers );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based interface index on which the server will be running.
paDNSAddr	[IN] Array of IPv4 addresses of DNS servers to use.
NumServers	[IN] Number of DNS servers in array.

Table 10.2: IP_DHCPConfigDNSAddr() parameter list

Return value

0: O.K.

Other: Error.

Additional information

Configuring DNS server settings is optional. If no DNS servers are configured no DNS servers will be assigned to clients.

Needs to be called before activating the DHCP server for this interface with *IP_DHCPStart()* on page 314.

Example

```
U32 aDNSAddr[2];

//
// Setup DNS addr. as needed.
//
aDNSAddr[0] = IP_BYTES2ADDR(192, 168, 12, 1);
aDNSAddr[1] = IP_BYTES2ADDR(192, 168, 12, 2);
IP_DHCPConfigDNSAddr(0, &aDNSAddr[0], 2);
IP_DHCPConfigPool(0, IP_BYTES2ADDR(192, 168, 12, 11), 0xFFFF0000, 20);
IP_DHCPInit(0);
IP_DHCPStart(0);
```

10.2.2 IP_DHCPConfigGWAddr()

Description

Configures the gateway addr. that will be assign to clients.

Prototype

```
int IP_DHCPConfigGWAddr ( unsigned IFIndex,
                          U32      GWAddr );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based interface index on which the server will be running.
GWAddr	[IN] IP addr. of gateway.

Table 10.3: IP_DHCPConfigGWAddr() parameter list

Return value

0: O.K.

Other: Error.

Additional information

Configuring a gateway setting is optional. If no gateway is configured no gateway will be assigned to clients.

Needs to be called before activating the DHCP server for this interface with *IP_DHCPStart()* on page 314.

Example

```
IP_DHCPConfigGWAddr(0, IP_BYTES2ADDR(192, 168, 12, 1));
IP_DHCPConfigPool(0, IP_BYTES2ADDR(192, 168, 12, 11), 0xFFFF0000, 20);
IP_DHCPInit(0);
IP_DHCPStart(0);
```

10.2.3 IP_DHCPConfigMaxLeaseTime()

Description

Configures the maximum lease time that a client will be granted to use the achieved configuration.

Prototype

```
int IP_DHCPConfigMaxLeaseTime ( unsigned IFIndex,  
                                U32      Seconds );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based interface index on which the server will be running.
Seconds	[IN] Maximum lease time in seconds. Default 7200s => 2h.

Table 10.4: IP_DHCPConfigMaxLeaseTime() parameter list

Return value

0: O.K.

Other: Error.

Additional information

Optional. Needs to be called before activating the DHCP server for this interface with *IP_DHCPStart()* on page 314.

Example

```
IP_DHCPConfigMaxLeaseTime(0, 7200);  
IP_DHCPConfigPool(0, IP_BYTES2ADDR(192, 168, 12, 11), 0xFFFF0000, 20);  
IP_DHCPInit(0);  
IP_DHCPStart(0);
```

10.2.4 IP_DHCPConfigPool()

Description

Configures the IP address pool that can be assigned to DHCP clients.

Prototype

```
int IP_DHCPConfigPool ( unsigned IFIndex,
                        U32      StartIPAddr,
                        U32      SNMask,
                        U32      PoolSize );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based interface index on which the server will be running.
StartIPAddr	[IN] First IP addr. of the pool.
SNMask	[IN] Subnet mask to assign to clients.
PoolSize	[IN] Number of IP addresses in pool starting from StartIPAddr .

Table 10.5: IP_DHCPConfigPool() parameter list

Return value

0: O.K.

Other: Error.

Additional information

Needs to be called before activating the DHCP server for this interface with *IP_DHCPStart()* on page 314.

Example

```
IP_DHCPConfigPool(0, IP_BYTES2ADDR(192, 168, 12, 11), 0xFFFF0000, 20);
IP_DHCPInit(0);
IP_DHCPStart(0);
```

10.2.5 IP_DHCPS_Halt()

Description

Halts the DHCP server on a specific interface.

Prototype

```
void IP_DHCPS_Halt ( unsigned IFIndex );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based interface index on which the server is running.

Table 10.6: IP_DHCPS_Halt() parameter list

10.2.6 IP_DHCP_Init()

Description

Initializes a DHCP server instance for an interface.

Prototype

```
int IP_DHCP_Init ( unsigned IFIndex );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based interface index on which the server will be running.

Table 10.7: IP_DHCP_Init() parameter list

Return value

0: O.K.

Other: Error.

Additional information

Needs to be called before activating the DHCP server for this interface with *IP_DHCP_Start()* on page 314.

10.2.7 IP_DHCPStart()

Description

Starts a DHCP server instance on an interface.

Prototype

```
int IP_DHCPStart ( unsigned IFIndex );
```

Parameter

Parameter	Description
IFIndex	[IN] Zero-based interface index on which the server will be running.

Table 10.8: IP_DHCPStart() parameter list

Return value

0: O.K.

Other: Error.

Additional information

IP_DHCPStart() on page 313 and *IP_DHCPConfigPool()* on page 311 need to be called before activating the DHCP server for an interface in order to set at least the minimum configurations.

10.3 DHCP server resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the DHCP server 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.

10.3.1 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 DHCP server	approximately 2.0Kbyte

Table 10.9: DHCP server ROM usage ARM7

10.3.2 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 DHCP server	approximately 2.0Kbyte

Table 10.10: DHCP server ROM usage Cortex-M3

10.3.3 RAM usage

Addon	RAM
embOS/IP DHCP server	approximately 200 bytes

Table 10.11: DHCP server RAM usage

Chapter 11

AutoIP

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

11.1 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.

11.2 API functions

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

Table 11.1: embOS/IP AutoIP interface function overview

11.2.1 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 11.2: 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 801 and *IP_AddLogFilter()* on page 799 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.
```


11.2.2 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 11.3: 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.

11.2.3 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
<code>IFaceId</code>	[IN] Zero-based interface index.
<code>pfInformUser</code>	[IN] Pointer to a user function of type <code>IP_AUTOIP_INFORM_USER_FUNC</code> which is called when a status change occurs.

Table 11.4: 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);
```

11.2.4 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 11.5: 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.

11.3 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.

11.3.1 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 11.6: AutoIP ROM usage ARM7

11.3.2 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 11.7: AutoIP ROM usage Cortex-M3

11.3.3 RAM usage

Addon	RAM
embOS/IP AutoIP module	approximately 0.7Kbyte

Table 11.8: AutoIP RAM usage

Chapter 12

Address Collision Detection

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

12.1 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.

12.2 API functions

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

Table 12.1: embOS/IP ACD interface function overview

12.2.1 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 12.2: 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 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 329 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 801 and *IP_AddLogFilter()* on page 799 for further information about the enabling terminal I/O.

12.2.2 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 12.3: IP_ACD_Config() parameter list

12.3 ACD data structures

12.3.1 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 12.4: Structure ACD_FUNC member list

12.4 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.

12.4.1 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 12.5: ACD ROM usage ARM7

12.4.2 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 12.6: ACD ROM usage Cortex-M3

12.4.3 RAM usage

Addon	RAM
embOS/IP ACD module	approximately 50Bytes

Table 12.7: ACD RAM usage

Chapter 13

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.

13.1 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 implement 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, <code>IP_UPnP.c</code> .

Supplied directory structure of embOS/IP UPnP package

13.2 Feature list

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

13.3 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.

13.4 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 distribute services provided by a device.

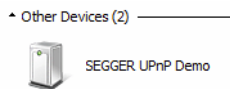
UPnP can be used to advertise services provided by a device across the network such as where to find the web interface for the device advertising. Newer operating systems support UPnP from scratch and will show UPnP devices available across a network and may provide easy access to a device by simply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

13.4.1 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_Webserver_UPnP.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_Webserver_UPnP.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_Webserver_UPnP.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 525 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_Webserver_UPnP.c */
/*****
*
*      Types
*
*****/

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

```

        unsigned    NumBytes;
    } VFILE_LIST;

/* Excerpt from OS_IP_Webserver_UPnP.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_Webserver_UPnP.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_Webserver_UPnP.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_Webserver_UPnP.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", 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:95232DE0-3AF7-11E2-81C1-"
    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 * pContextOut, 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"
        "<URLBase>"
        _UPnP_GetURLBase(ac, sizeof(ac))
        "</URLBase>"
        , 0);
}

```

```

, 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"
                                "<SCPDUURL>/dummy.xml</SCPDUURL>\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_Webserver_UPnP.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_COUNT_OF(_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_Webserver_UPnP.c */
//
// Activate UPnP with VFile hook for needed XML files
//
IP_WEBS_AddVFileHook(&_UPnP_VFileHook, &_UPnP_VFileAPI);
IP_UPNP_Activate(INTERFACE, NULL);

```


13.5 API functions

Function	Description
<code>IP_UPNP_Activate()</code>	Activates UPnP advertisement of the target in the network.

Table 13.1: embOS/IP UPnP API function overview

13.5.1 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 13.2: IP_UPNP_Activate() parameter list

Additional information

If [acUDN](#) is NULL the unique descriptor name will be generated from the HW addr. of the interface.

13.6 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.

13.6.1 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 13.3: UPnP ROM usage ARM7

13.6.2 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 13.4: UPnP ROM usage Cortex-M3

13.6.3 RAM usage

Addon	RAM
embOS/IP UPnP	approximately 170 bytes

Table 13.5: UPnP RAM usage

Chapter 14

VLAN

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

14.1 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.

14.2 Feature list

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

14.3 VLAN backgrounds

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

A typical usage would be to have 2 departments separated 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
00:23:C7:FF:FF:FF	00:23:C7:FF:EE:EE	TPI	16 bit TCI (12 bit VLAN ID)	IP Packet 0x0800	Max. 1496 bytes data + 4 bytes CRC
		0x8100	VLAN ID #2 0x0002		

Ethernet frame of max. 1518 bytes

14.4 API functions

Function	Description
<code>IP_VLAN_AddInterface()</code>	Activates UPnP advertisement of the target in the network.

Table 14.1: embOS/IP VLAN API function overview

14.4.1 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 14.2: IP_VLAN_AddInterface() parameter list

Return value

Zero-based index of the added VALN interface.

14.5 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.

14.5.1 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 14.3: VLAN ROM usage ARM7

14.5.2 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 14.4: VLAN ROM usage Cortex-M3

14.5.3 RAM usage

Addon	RAM
embOS/IP VLAN	approximately 16 bytes

Table 14.5: VLAN RAM usage

Chapter 15

Tail Tagging (Add-on)

The embOS/IP support for the Micrel Tail Tagging feature that is available in many Micrel Switch PHYs is an optional extension to embOS/IP. It can be used to extend a typical single Ethernet port CPU with more full featured ports without having to redesign a complete hardware or even changing to a completely other CPU with more Ethernet ports. This chapter contains information about Tail Tagging and how to add it to your hardware and software.

15.1 embOS/IP Tail Tagging support

The embOS/IP Tail Tagging implementation is an optional extension which can be easily added to extend your hardware using a Micrel Switch PHY with Tail Tagging support instead of a single port PHY. It allows you to extend your single Ethernet port (also single Ethernet controller) CPU to as many ports that can be managed like a real network interface (Ethernet controller) in embOS/IP even with different hardware addresses.

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

Directory	Content
BSP	Contains sample configurations for hardware that already uses embOS/IP Tail Tagging support.
IP	Contains the Tail Tagging sources, IP_MICREL_TAIL_TAGGING.c and the PHY driver for various Micrel Switch PHYs IP_PHY_MICREL_SWITCH.c.

Supplied directory structure of embOS/IP Tail Tagging package

15.2 Feature list

- Extend virtually any single port CPU to n manageable interfaces at low cost.
- Use the fast MII/RMII interface of your CPU and internal Ethernet controller instead of slower interfaces like SPI with external Ethernet controllers.
- Link status of each port can be monitored independent.
- Keep your existing design and known and preferred CPU.
- Each Tail Tagging interface can have its own hardware address.
- Low memory footprint.
- Seamless integration with the embOS/IP stack.

15.3 Use cases for Tail Tagging

The benefits of Tail Tagging are that it can be used to extend a single port Ethernet CPU to multiple, manageable physical ports where each port can be managed independently and can even have its own hardware address assign.

This can be used for various purposes when bulding hardware and software with special requirements. Some use cases are:

- Bulding a mult homing hardware that shall be fail safe on the network by providing multiple network paths that at the same time shall act as completely independent interfaces with full control.
- Building a low cost Router, Gateway or Bridge device interfacing multiple networks.
- Building a device that requires network separation features and at the same time is still able to use other techniques like VLAN/prioritizing via VLAN. VLAN can be used in a similar way than Tail Tagging but can not provide both features (VLAN and port separation) at the same time.

15.4 Requirements

The following requirements regarding software and hardware need to be met.

15.4.1 Software requirements

The embOS/IP Tail Tagging implementation requires the embOS/IP TCP/IP stack and a PHY driver for a Micrel Switch PHY that supports the Tail Tagging feature.

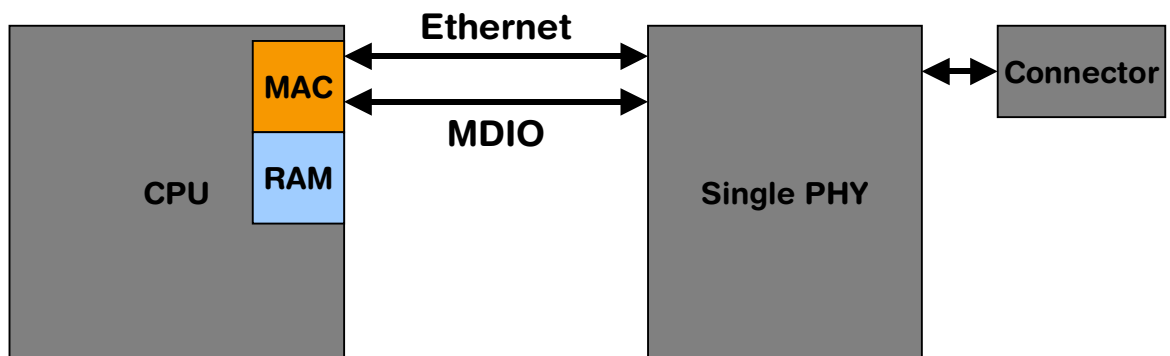
15.4.2 Hardware requirements

Of course a Micrel Switch PHY supporting Tail Tagging needs to be present on your hardware as well. The big advantage of using Tail Tagging instead of other methods like adding external Ethernet controllers is its simplicity that comes without any known downsides.

Single MAC unit CPU, single port design

The typical hardware design for an Ethernet capable hardware with the MAC unit inside the CPU is shown below. It consists of a CPU with a single internal MAC unit connected to an external single port PHY that can interface one port to the network.

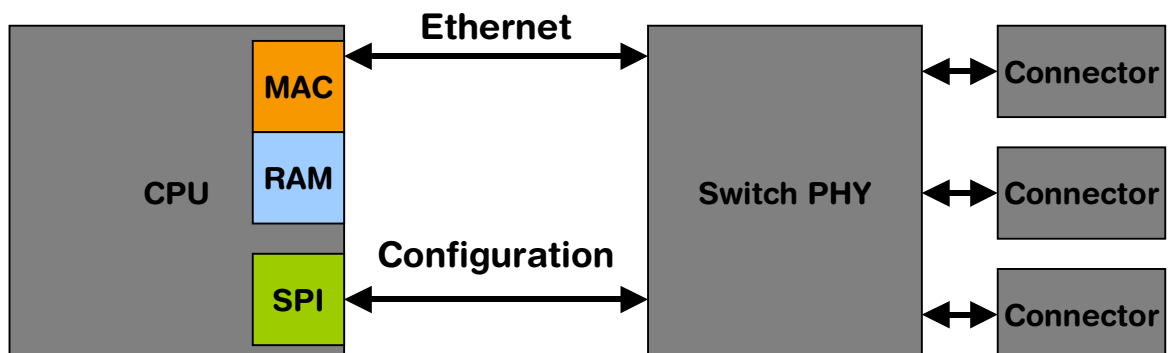
The Ethernet data is transferred between MAC and PHY while the MDIO interface (typically also accessed via registers of the MAC) is used to access the PHY to configure it and periodically check the link status.



Single MAC unit CPU, switch PHY with Tail Tagging design

For Tail Tagging only a few simple changes to the hardware are necessary. The main difference is that configuration is no longer done via the MDIO interface but instead is done using an extra interface like SPI or SMI. This is due to a restricted set of registers that are available via the MDIO standard.

Typically the same registers that can be accessed via MDIO can be accessed via SPI or SMI as well, along many other registers not available via MDIO.



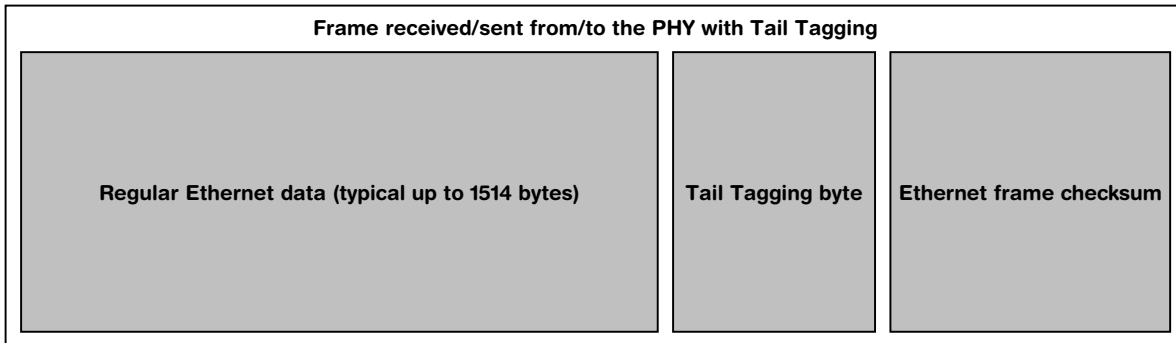
Using a Switch PHY with Tail Tagging not only allows you to connect multiple hosts but also allows you to fully control each external port/connector like it would be an additional expensive and external Ethernet controller.

15.5 Tail Tagging backgrounds

The Tail Tagging feature available in many Micrel Switch PHYs is a clever way to pass information between the PHY and the TCP/IP stack on which port of the Switch a packet has been received or to which port(s) it should be delivered when the TCP/IP stack sends data to the network.

Contents of a Tail Tagging frame

The picture below shows the content of a frame that is received from the Switch in the host or is sent from the host to the Switch.



When the Switch has the Tail Tagging feature enabled all ports of the Switch will be used in this mode.

Receiving a frame with Tail Tagging

With Tail Tagging each Ethernet frame that is received will be added with a byte between the Ethernet data received in the frame and the checksum of the Ethernet frame itself. This step is unseen by the Ethernet controller as the frame checksum that is built by the sender above all the Ethernet data in the frame is altered by the PHY as well to represent the correct checksum of the original Ethernet data in the frame plus the byte that has been added. Due to the correct checksum the Ethernet controller does not have to be aware of Tail Tagging at all.

embOS/IP can then extract the information from which port the data has been received from the Tail Tagging byte and can assign the packet to the correct Tail Tagging interface in the system. The Tail Tagging byte is stripped in this process leaving only the original data that can then be transferred to upper layer protocols.

Sending a frame with Tail Tagging

Sending works similar than receiving a frame. Before the Ethernet frame is queued with the Ethernet controller for transmitting it to the PHY, a Tail Tagging byte is appended at the end of the data to send (and before the frame checksum is calculated and added by the Ethernet driver itself). This byte contains the information to which external PHY ports the packet shall be delivered and sent out to the network.

The whole process is again unseen by the Ethernet controller as it is only aware that the data to be sent is one byte more in total like if one byte more would be sent by an upper layer protocol.

15.6 Optimal MTU and buffer sizes

A Tail Tagging interface in embOS/IP is a virtual interface that uses a hardware interface for data transfer. As Tail Tagging requires to store one additional byte that is unknown to upper layer protocols the Tail Tagging byte is automatically subtracted from the MTU that has been configured for the hardware interface.

While simply using the original MTU - 1 is a safe and easy way it has the downside that the maximum MTU of an Ethernet packet is now 1499 bytes instead of 1500 bytes and might lead to slight fragmentation and small delays with hardware and other hosts that are optimized for MTUs of 1500 bytes.

To overcome this effect the MTU (and typically connected with it the size of the big packet buffers) in `IP_X_Config()` should not be configured to 1500 bytes but instead configured to 1501 bytes if it is known that Tail Tagging will be used.

If a mix of Tail Tagging and non Tail Tagging interfaces will be used (dual Ethernet controller in CPU, one using only single port and the other connected to a Switch using Tail Tagging) the MTU should be set accordingly for each of these interfaces using `IP_SetMTU()` on page 99.

15.7 API functions

Function	Description
Tail Tagging	
<code>IP_MICREL_TAIL_TAGGING_AddInterface()</code>	Adds a virtual interface to the stack using the Micrel Tail Tagging feature to separate switch ports.

Table 15.1: embOS/IP Tail Tagging API function overview

15.7.1 IP_MICREL_TAIL_TAGGING_AddInterface()

Description

Adds a virtual interface to the stack using the Micrel Tail Tagging feature to separate switch ports.

Prototype

```
int IP_MICREL_TAIL_TAGGING_AddInterface( unsigned HWIFace,
                                         U8      InTag,
                                         U8      OutTag );
```

Parameter

Parameter	Description
HWIFaceId	Zero-based interface index of the interface used as hardware interface.
InTag	Tag byte according to Micrel documentation to compare with an incoming (switch to target) Tail Tagging byte.
OutTag	Tag byte according to Micrel documentation to append for an outgoing (target to switch) packet on this interface. Multiple bits can be set to allow sending to multiple ports at the same time.

Table 15.2: IP_MICREL_TAIL_TAGGING_AddInterface() parameter list

Return value

Interface index: ≥ 0

Error: < 0

Example

The following is an example based on an excerpt of the content from an IP_Config_*.c on how Tail Tagging can be added to embOS/IP:

```

/*****
 *
 *      Configuration
 *
 *****/

#define ALLOC_SIZE    0x6000                // Size of memory dedicated to
                                           // the stack in bytes.

#define DRIVER        &IP_Driver_K64      // Driver used for target.
#define TARGET_NAME    "emPowerV2_1"       // Target name used for DHCP
                                           // client.

#define HW_ADDR        "\x00\x22\xC7\xFF\xFF\x22" // MAC addr. used for target.
#define NUM_PORTS      3                  // Number of switch ports not
                                           // connected to the host CPU.

/*****
 *
 *      _ReadSPIReg()
 *
 *      Function description
 *      Reads a byte from a register.
 *
 *      Parameters
 *      pContext: Context of the PHY driver.
 *      Reg      : Address of the register to read.
 *
 *      Return value
 *      value read from register.
 */
static unsigned _ReadSPIReg(IP_PHY_CONTEXT_EX* pContext, unsigned Reg) {
    U8 v;

    IP_USE_PARA(pContext);

    v = READ_REG(Reg);
    return v;
}

/*****
 *
 *      _WriteSPIReg()
 *
 *      Function description
 *      Writes a byte to a register.
 *
 *      Parameters
 *      pContext: Context of the PHY driver.
 *      Reg      : Address of the register to read.
 *      v        : Data to write.
 */
static void _WriteSPIReg(IP_PHY_CONTEXT_EX* pContext, unsigned Reg, unsigned v) {
    IP_USE_PARA(pContext);

    WRITE_REG(Reg, v);
}

/*****
 *
 *      _ConfigPHY()
 *
 *      Function description

```

```

*   Callback executed during the PHY init of the stack to configure
*   PHY settings once the hardware interface has been initialized.
*
*   Parameters
*   IFaceId: Zero-based interface index.
*/
static void _ConfigPHY(unsigned IFaceId) {
    if (IFaceId > 0) { // Exclude the hardware interface.
        //
        // Activate Tail Tagging. Needs to be done for one port of the switch
        // only as Tail Tagging is globally enabled for all ports but does
        // not hurt doing it again.
        //
        IP_PHY_MICREL_SWITCH_ConfigTailTagging(IFaceId, 1); // 0: Off, 1: On.
        //
        // Configure the physical zero-based port number on the switch for
        // this interface.
        // In this sample the port number is always one lower than the interface ID.
        //
        IP_PHY_MICREL_SWITCH_AssignPortNumber(IFaceId, IFaceId - 1);
    }
}

/*****
*
*   Local API structures
*
*****/

static const IP_PHY_MICREL_SWITCH_ACCESS PhyAccess = {
    _ReadSPIReg, // pfReadReg
    _WriteSPIReg // pfWriteReg
};

/*****
*
*   IP_X_Config()
*
*****/
void IP_X_Config(void) {
    int mtu;
    int IFaceId;
    int HWIFaceId;
    int i;
    U8 abHWAddr[6];

    _InitPhyIF(); // Initialize the interface for
                  // the switch configuration.

    IP_AssignMemory(_aPool, sizeof(_aPool)); // Assigning memory should be
                                              // the first thing.

    HWIFaceId = IP_AddEtherInterface(DRIVER); // Add driver for your hardware.
    IP_NI_ConfigPHYMode(HWIFaceId, 1); // Configure PHY Mode: 0: MII,
                                      // 1: RMII; For required hardware
                                      // changes for RMII, please refer
                                      // to your board manual.

    //
    // Define log and warn filter.
    // Note: The terminal I/O emulation might affect the timing of your
    // application, since most debuggers need to stop the target
    // for every terminal I/O output unless you use another
    // implementation such as DCC or SWO.
    //
    IP_SetWarnFilter(0xFFFFFFFF); // 0xFFFFFFFF: Do not filter: Output
                                // all warnings.

    IP_SetLogFilter(0
    ...

```

```

    );
//
// Add protocols to the stack (that do not require an interface parameter).
//
IP_TCP_Add();
IP_UDP_Add();
IP_ICMP_Add();
//
// Run-time configuration that needs to be set as it will
// be passed to virtual interfaces from the HW interface.
//
mtu = 1500;                // 576 is minimum acc. to RFC, 1500 is max.
                           // for Ethernet.
IP_SetMTU(HWIFaceId, mtu); // Maximum Transmission Unit is 1500 for
                           // Ethernet by default.

//
// Configure each switch port (not connected to the host CPU).
//
IP_MEMCPY(&abHWAddr[0], (const U8*)HW_ADDR, 6);
for (i = 0; i < NUM_PORTS; i++) {
    //
    // Add Tail Tagging interface for switch port.
    //
    IFaceId = IP_MICREL_TAIL_TAGGING_AddInterface(HWIFaceId, i, (1 < i));
    //
    // Set MAC addr. for switch port: Needs to be unique for production units.
    //
    IP_SetHWAddrEx(IFaceId, (const U8*)&abHWAddr[0], 6);
    //
    // Increase last byte of HW addr. for next switch port.
    //
    abHWAddr[5]++;
    //
    // Add PHY driver for Micrel switch PHY to the interface.
    //
    IP_PHY_AddDriver(IFaceId,
                     &IP_PHY_Driver_Micrel_Switch_KSZ8895,
                     &PhyAccess,
                     &ConfigPHY);

    //
    // Activate DHCP client for this interface.
    //
    IP_DHCPC_Activate(IFaceId, TARGET_NAME, NULL, NULL);
    //
    // Add IPv6 support to the stack and enable it for the interface.
    //
#ifdef IP_SUPPORT_IPV6
    IP_IPV6_Add(IFaceId);
#endif
}
//
// Run-time configure buffers.
// The default setup will do for most cases.
//
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 the TCP Tx and Rx
                                                    // window size

IP_SOCKET_SetDefaultOptions(0
                             ...
                             );
}

```


15.8 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the Tail Tagging module presented in the tables below have been measured on a Cortex-M4 system. Details about the further configuration can be found in the sections of the specific example.

15.8.1 ROM usage on a Cortex-M4 system

The following resource usage has been measured on a Cortex-M4 system using SEGGER Embedded Studio V2.12, size optimization.

Addon	ROM
embOS/IP Tail Tagging module	approximately 0.4KBytes

Table 15.3: Tail Tagging ROM usage Cortex-M4

15.8.2 RAM usage

All required RAM is taken from the RAM that has been assigned to embOS/IP using *IP_AddMemory()* on page 57. Only a few bytes are required.

Chapter 16

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.

16.1 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 402.

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.

16.1.1 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.

16.1.2 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 indicate this to the stack. In this case, the stack does not compute these checksums, improving throughput and reducing CPU load.

16.1.3 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.

16.2 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 16.1: 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 54 for detailed information.

16.2.1 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.

16.2.1.1 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 16.2: List of tested eval boards

16.2.1.2 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 54 and *IP_X_Configure()* on page 422 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 RMI 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_DHCP_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);
}
```


16.2.1.3 Driver specific configuration functions

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

Table 16.3: embOS/IP CAP9 driver specific function overview

16.2.1.3.1 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 16.4: IP_NI_CAP9_ConfigNumRxBuffers() parameter list

16.2.1.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 16.5: embOS/IP driver specific function overview

16.2.1.4.1 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 16.6: 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
}
```

16.2.1.5 Additional information

None.

16.2.2 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.

16.2.2.1 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 16.7: List of tested eval boards

16.2.2.2 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 54 and *IP_X_Configure()* on page 422 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_DHCP_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);
}

```

16.2.2.3 Driver specific configuration functions

Function	Description
IP_NI_AT91RM9200_ConfigNumRxBuffers()	Sets the number of Rx buffers.

Table 16.8: embOS/IP RM9200 driver specific function overview

16.2.2.3.1 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 16.9: IP_NI_RM9200_ConfigNumRxBuffers() parameter list

16.2.2.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 16.10: embOS/IP driver specific function overview

16.2.2.4.1 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 16.11: 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
}
```

16.2.2.5 Additional information

None.

16.2.3 ATMEL AT91SAM7X

The ATMEL AT91SAM7X's are flash microcontrollers with integrated Ethernet, USB and CAN interfaces, based on the 32-bit ARM7TDMI RISC processor.

16.2.3.1 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 16.12: List of tested eval boards

16.2.3.2 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 54 and *IP_X_Configure()* on page 422 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_DHCP_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);
}
```

16.2.3.3 Driver specific configuration functions

Function	Description
IP_NI_SAM7X_ConfigNumRxBuffers()	Sets the number of Rx buffers.

Table 16.13: embOS/IP SAM7X driver specific function overview

16.2.3.3.1 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 16.14: IP_NI_SAM7X_ConfigNumRxBuffers() parameter list

16.2.3.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 16.15: embOS/IP driver specific function overview

16.2.3.4.1 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 16.16: 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
}

```

16.2.3.5 Additional information

None.

16.2.4 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.

16.2.4.1 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 16.17: List of tested eval boards

16.2.4.2 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 54 and *IP_X_Configure()* on page 422 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_DHCP_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);
}
```

16.2.4.3 Driver specific configuration functions

Function	Description
<code>IP_NI_SAM9260_ConfigNumRxBuffers()</code>	Sets the number of Rx buffers.

Table 16.18: embOS/IP SAM9260 driver specific function overview

16.2.4.3.1 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 16.19: IP_NI_SAM9260_ConfigNumRxBuffers() parameter list

16.2.4.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 16.20: embOS/IP driver specific function overview

16.2.4.4.1 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 16.21: 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

#ifdef _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)
}

```

16.2.4.5 Additional information

None.

16.2.5 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.

16.2.5.1 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 16.22: List of tested eval boards

16.2.5.2 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 54 and *IP_X_Configure()* on page 422 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_DHCP_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 );
}
```

16.2.5.3 Driver-specific configuration functions

Function	Description
<code>IP_NI_DM9000_ConfigAddr()</code>	Sets the base address for commands and data register.
<code>IP_NI_DM9000_ISR_Handler()</code>	Interrupt service routine for the network interface.

Table 16.23: embOS/IP DM9000 driver-specific function overview

16.2.5.3.1 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
<code>Unit</code>	[IN] Zero-based index of available network interfaces.
<code>pBase</code>	[IN] Pointer to the control register of the MAC.
<code>pValue</code>	[IN] Pointer to the data register of the MAC.

Table 16.24: 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 422 for detailed information.

16.2.5.3.2 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
<code>Unit</code>	[IN] Zero-based index of available network interfaces.

Table 16.25: IP_NI_DM9000_ISR_Handler() parameter list

16.2.5.4 Required BSP functions

Function	Description
<code>BSP_ETH_Init()</code>	Initializes the network interface.

Table 16.26: embOS/IP driver specific function overview

16.2.5.4.1 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 );
```

16.2.5.5 Additional information

None.

16.2.6 FREESCALE ColdFire MCF5329

16.2.6.1 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 COLD FIRE SDK with MCF5329 Fire Engine

Table 16.27: List of tested eval boards

16.2.6.2 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 54 and *IP_X_Configure()* on page 422 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_DHCP_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  
    );  
}
```

16.2.6.3 Driver-specific configuration functions

None.

16.2.6.4 Required BSP functions

None.

16.2.6.5 Additional information

None.

Parameter

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

Table 16.28: 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
    }
}

```

16.2.6.6 Additional information

None.

16.2.7 NXP LPC17xx

The NXP LPC17xx MCUs are flash microcontrollers with integrated Ethernet, USB and CAN interfaces, based on the 32-bit Cortex-M3 processor.

16.2.7.1 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 16.29: List of tested eval boards

16.2.7.2 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 54 and *IP_X_Configure()* on page 422 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_DHCP_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
                    );
}
```

16.2.7.3 Driver-specific configuration functions

None.

16.2.7.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 16.30: embOS/IP driver specific function overview

16.2.7.4.1 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 16.31: 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;
}
```

16.2.7.5 Additional information

None.

16.2.8 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.

16.2.8.1 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 16.32: List of tested eval boards

16.2.8.2 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 54 and *IP_X_Configure()* on page 422 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_DHCP_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
                    );
}
```

16.2.8.3 Driver-specific configuration functions

None.

16.2.8.4 Required BSP functions

Function	Description
BSP_ETH_Init()	Initializes the network interface.

Table 16.33: embOS/IP driver specific function overview

16.2.8.4.1 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 16.34: 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;
}
```

16.2.8.5 Additional information

None.

16.2.9 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.

16.2.9.1 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 16.35: List of tested eval boards

16.2.9.2 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 54 and *IP_X_Configure()* on page 422 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_DHCP_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);
}
```

16.2.9.3 Driver-specific configuration functions

None.

16.2.9.4 Required BSP functions

None.

16.2.9.5 Additional information

None.

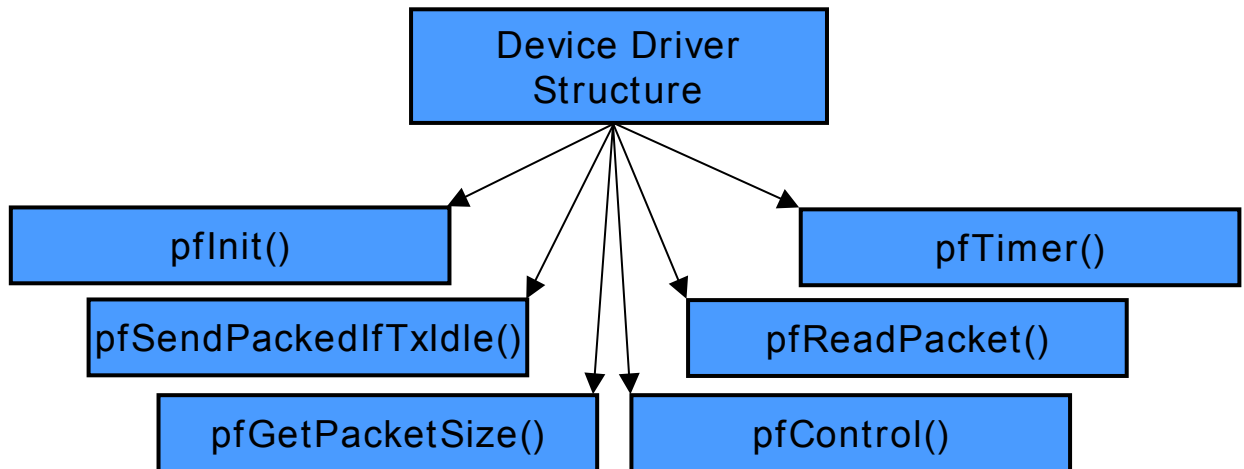
16.3 Writing your own driver

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.

16.3.1 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
<code>pfInit</code>	Pointer to the initialization function.
<code>pfSendPacket</code>	Pointer to the send packet function.
<code>pfGetPacketSize</code>	Pointer to the get packet size function.
<code>pfReadPacket</code>	Pointer to the read packet function.
<code>pfTimer</code>	Optional: Pointer to the timer function. The routine is called from the stack periodically.
<code>pfControl</code>	Pointer to the control function.

Table 16.36: 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
};
  
```

16.3.2 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
<code>pfControl()</code>	This function is used to implement additional driver specific control functions. It can be empty.
<code>pfInit()</code>	General initialization function of the driver.
<code>pfGetPacketSize()</code>	Reads buffer descriptors to find out if a packet has been received.
<code>pfReadPacket()</code>	Reads the first packet in the buffer.
<code>pfSendPacketIfTxIdle()</code>	Send the next packet in the send queue if transmitter is idle.
<code>pfTimer()</code>	Timer function called by the networking task, <code>IP_Task()</code> , once per second.

Table 16.37: embOS/IP network interface driver functions

16.3.3 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.
*      Alternatively, 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 capabilities, 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
// | IP_NI_CAPS_WRITE_TCP_CHKSUM // UDP-checksum into an outgoing packet?
// | IP_NI_CAPS_WRITE_ICMP_CHKSUM // Driver capable of inserting the
// | IP_NI_CAPS_CHECK_IP_CHKSUM // TCP-checksum into an outgoing packet?
// | IP_NI_CAPS_CHECK_UDP_CHKSUM // Driver capable of inserting the
// | IP_NI_CAPS_CHECK_TCP_CHKSUM // ICMP-checksum into an outgoing packet?
// | IP_NI_CAPS_CHECK_ICMP_CHKSUM // Driver capable of computing and
// | IP_NI_CAPS_CHECK_IP_CHKSUM // comparing the IP-checksum of
// | IP_NI_CAPS_CHECK_UDP_CHKSUM // incoming packets?
// | IP_NI_CAPS_CHECK_TCP_CHKSUM // Driver capable of computing and
// | IP_NI_CAPS_CHECK_ICMP_CHKSUM // comparing the UDP-checksum of an
// | IP_NI_CAPS_CHECK_IP_CHKSUM // incoming packet?
// | IP_NI_CAPS_CHECK_UDP_CHKSUM // Driver capable of computing
// | IP_NI_CAPS_CHECK_TCP_CHKSUM // and comparing the TCP-checksum of
// | IP_NI_CAPS_CHECK_ICMP_CHKSUM // an incoming packet?
// | IP_NI_CAPS_CHECK_IP_CHKSUM // Driver capable of computing
// | IP_NI_CAPS_CHECK_UDP_CHKSUM // and comparing the ICMP-checksum of
// | IP_NI_CAPS_CHECK_TCP_CHKSUM // an incoming packet?
// | IP_NI_CAPS_CHECK_ICMP_CHKSUM // Driver capable of computing
// | IP_NI_CAPS_CHECK_IP_CHKSUM // and comparing the ICMP-checksum of
// | IP_NI_CAPS_CHECK_UDP_CHKSUM // an incoming packet?
// | IP_NI_CAPS_CHECK_TCP_CHKSUM // Driver capable of computing
// | IP_NI_CAPS_CHECK_ICMP_CHKSUM // and comparing the ICMP-checksum of
// | IP_NI_CAPS_CHECK_IP_CHKSUM // 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 17

PHY drivers

embOS/IP has been designed to cooperate with any kind of hardware. Typically almost any PHY is compatible with the embOS/IP generic PHY driver as almost all standard PHYs are compliant to the `IEEE 802.3u` standard which also defines the first 6 standard registers and their bits that are used by the generic PHY driver. However there are some PHYs that might require additional setup or do not comply with IEEE 802.3u as it is expected by the generic PHY driver. To use them, a so-called PHY driver for that hardware is required that is aware of how the specific PHY can be accessed.

17.1 General information

To use embOS/IP with a PHY that can not be used with the generic PHY driver, a specific PHY driver matching the target hardware is required. The code size of a PHY driver depends on the hardware but is typically only requires a couple of hundred bytes.

The PHY driver interface has been designed to allow support of generic features like checking the link state as well as being able to extend each specific driver with unique features only available for a specific hardware.

17.1.1 When is a specific PHY driver required?

A specific PHY driver is typically not required for any standard PHY on the market. However it might be required for the following reasons:

- The PHY registers do not (fully) comply with the IEEE 802.3u standard for the six first registers.
- The PHY requires additional setup that can not be provided using the reset hook of the generic PHY driver.
- A PHY or (managed) Switch PHY shall be used that includes multiple PHYs that shall be treated like autonomous PHYs for link checking on each port.
- Any other special solution that simply can not be covered by a generic PHY driver.

17.2 Available PHY drivers

embOS/IP comes with a generic PHY driver that fits virtually any standard single port PHY that is on the market. PHY drivers for devices that are not compatible to the IEEE 802.3u standard, require additional setup or special solutions are optional components to embOS/IP. A list of available PHY drivers not including the generic PHY driver that always comes with embOS/IP can be found at the following location:

The following PHY drivers are described in detail with their available API:

PHY driver	Identifier
Generic driver	IP_PHY_Driver_Generic
Micrel Switch PHY driver	IP_PHY_Driver_Micrel_Switch_<Product Name>

Table 17.1: List of PHY drivers and their labels

To add a PHY driver to embOS/IP, `IP_PHY_AddDriver()` should be called from within `IP_X_Config()` with the proper identifier. Refer to *IP_PHY_AddDriver()* on page 130 for detailed information.

17.2.1 Generic driver

The embOS/IP generic PHY driver fits virtually any standard single port PHY that complies with the IEEE 802.3u standard and is the default driver that comes with embOS/IP and is used when no other PHY driver is added for an interface that requires PHY support.

Warning: Even if a PHY complies with the IEEE 802.3u standard it might require additional handling that can not be provided by the generic PHY driver and therefore might require a specific PHY driver in this case.

Resource usage

The following resource usage has been measured on a Cortex-M4 system using SEGGER Embedded Studio V2.12, size optimization.

ROM	RAM
approximately 0.8KBytes	0KBytes

Table 17.2: embOS/IP generic PHY driver resource usage Cortex-M4

All required RAM is taken from the RAM that has been assigned to embOS/IP using *IP_AddMemory()* on page 57. Only a few bytes are required.

17.2.1.1 Generic PHY driver API functions

The table below lists the available API functions for this driver:

Function	Description
IP_PHY_GENERIC_AddResetHook()	Adds a reset hook for custom init.
IP_PHY_GENERIC_RemapAccess()	Allows using the access API of one interface with another.

Table 17.3: embOS/IP generic PHY driver API function overview

17.2.1.2 IP_PHY_GENERIC_AddResetHook()

Description

Adds a hook to a callback that is executed each time after the stack has applied a software reset to the PHY.

Prototype

```
void IP_PHY_GENERIC_AddResetHook (          IP_HOOK_ON_PHY_RESET *pHook,
                                         void (*pf) (          unsigned IFaceId,
                                         void *pContext,
                                         const IP_PHY_API *pApi) );
```

Parameter

Parameter	Description
pHook	[IN] Pointer to static element of IP_HOOK_ON_PHY_RESET that can be internally used by the stack.
pf	[IN] Function pointer to the callback that will be executed.

Table 17.4: IP_PHY_GENERIC_AddResetHook() parameter list

Additional information

In some cases it might be necessary to apply a custom configuration to the PHY. The generic PHY module used by the stack in most cases will only apply a minimal configuration. Registering a callback custom settings can be applied to this configuration.

If you are changing the PHY register page you need to reset it back to page 0 before returning from the callback.

Example

```
//
// Excerpt of content of IP_Config_*.c
//
static IP_HOOK_ON_PHY_RESET _Hook;

/*****
 *
 *      _OnPhyReset()
 *
 * Function description
 *      Callback called after a PHY reset and generic initialization has
 *      been applied by the stack to allow the user to apply his own
 *      settings if necessary.
 *
 * Parameters
 *      IFaceId : Zero-based interface ID.
 *      pContext: PHY context.
 *      pApi     : PHY access API.
 */
static void _OnPhyReset(unsigned IFaceId, void *pContext, const IP_PHY_API *pApi) {
    U16 v;

    v = pApi->pfRead(pContext, 0); // Read PHY register 0.
    ...                          // Modify value read.
    pApi->pfWrite(pContext, 0, v); // Write modified value back to PHY register 0.
}

void IP_X_Config(void) {
    ...
    IP_PHY_GENERIC_AddResetHook(&_Hook, _OnPhyReset); // Register _OnPhyReset() to
                                                         // be executed after a PHY
                                                         // software reset.
    ...
}
```

17.2.1.3 IP_PHY_GENERIC_RemapAccess()

Description

This function allows remapping the access routines of a PHY interface. An example would be to use the access routines of interface #0 for interface #1 as well.

Prototype

```
void IP_PHY_GENERIC_RemapAccess( unsigned IFaceId,  
                                unsigned AccessIFaceId );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index to assign an access API.
AccessIFaceId	Zero-based interface index from where to use the access API.

Table 17.5: IP_PHY_GENERIC_RemapAccess() parameter list

Additional information

The purpose to use the same MDIO interface for multiple PHY interfaces is that there are dual PHYs out there like the [TI DP83849I](#) that use only one MDIO interface and address their internal dual PHY via the PHY addr.

It is only possible to remap from an already initialized interface to a new one which means [AccessIFaceId](#) needs to be higher than [IFaceId](#).

17.2.2 Micrel Switch PHY driver

Due to the nature of a Switch PHY it contains multiple ports that can not be handled in a generic way. Therefore a driver that is aware of the specific hardware is required. The embOS/IP PHY driver for Micrel Switch PHYs supports automatically starting the Switch engine (autostart is disabled if used with a management interface like SMI or SPI) and allows further specific configuration for various purposes.

Supported devices

The following is a list of supported devices and their labels:

PHY driver	Identifier
KSZ8794	IP_PHY_Driver_Micrel_Switch_KSZ8794
KSZ8895	IP_PHY_Driver_Micrel_Switch_KSZ8895

Table 17.6: List of supported Micrel Switch PHYs and their labels

Resource usage

The following resource usage has been measured on a Cortex-M4 system using SEGGER Embedded Studio V2.12, size optimization.

ROM	RAM
approximately 0.2KBytes	0KBytes

Table 17.7: embOS/IP Micrel Switch PHY driver resource usage Cortex-M4

All required RAM is taken from the RAM that has been assigned to embOS/IP using *IP_AddMemory()* on page 57. Only a few bytes are required.

17.2.2.1 Micrel Switch PHY driver API functions

The table below lists the available API functions for this driver:

Function	Description
IP_PHY_MICREL_SWITCH_AssignPortNumber()	Assigns a physical port to an interface.
IP_PHY_MICREL_SWITCH_ConfigLearnDisable()	Disable address learning on port.
IP_PHY_MICREL_SWITCH_ConfigRxEnable()	Enable/disable Rx on port.
IP_PHY_MICREL_SWITCH_ConfigTailTagging()	Configures Tail Tagging support.
IP_PHY_MICREL_SWITCH_ConfigTxEnable()	Enable/disable Tx on port.

Table 17.8: embOS/IP Micrel Switch PHY driver API function overview

17.2.2.2 IP_PHY_MICREL_SWITCH_AssignPortNumber()

Description

This function assigns the physical switch port number to an interface.

Prototype

```
void IP_PHY_MICREL_SWITCH_AssignPortNumber( unsigned IFaceId,  
                                             unsigned Port );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
Port	Zero-based physical port number on the switch.

Table 17.9: IP_PHY_MICREL_SWITCH_AssignPortNumber() parameter list

17.2.2.3 IP_PHY_MICREL_SWITCH_ConfigLearnDisable()

Description

This function can set the learn disable config for a specific port to enabled/disabled.

Prototype

```
void IP_PHY_MICREL_SWITCH_ConfigLearnDisable( unsigned IFaceId,
                                              unsigned OnOff );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	0: Switch addr. learning for the port enabled. 1: Switch addr. learning for the port disabled.

Table 17.10: IP_PHY_MICREL_SWITCH_ConfigLearnDisable() parameter list

17.2.2.4 IP_PHY_MICREL_SWITCH_ConfigRxEnable()

Description

This function can set Rx (network to switch) for a specific port to enabled/disabled.

Prototype

```
void IP_PHY_MICREL_SWITCH_ConfigRxEnable( unsigned IFaceId,  
                                           unsigned OnOff );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	0: Rx for the port disabled. 1: Rx for the port enabled.

Table 17.11: IP_PHY_MICREL_SWITCH_ConfigRxEnable() parameter list

17.2.2.5 IP_PHY_MICREL_SWITCH_ConfigTailTagging()

Description

This function switches Tail Tagging on/off.

Prototype

```
void IP_PHY_MICREL_SWITCH_ConfigTailTagging( unsigned IFaceId,
                                             unsigned OnOff ) ;
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	0: Tail Tagging off. 1: Tail Tagging on.

Table 17.12: IP_PHY_MICREL_SWITCH_AssignPortNumber() parameter list

Additional information

It is enough to set it for one port of the switch as the bit to change is in a register that is shared between all ports.

Tail Tagging needs to be supported by the stack as well and a Tail Tagging aware interface has to be added to the stack.

17.2.2.6 IP_PHY_MICREL_SWITCH_ConfigTxEnable()

Description

This function can set Tx (switch to network) for a specific port to enabled/disabled.

Prototype

```
void IP_PHY_MICREL_SWITCH_ConfigTxEnable( unsigned IFaceId,  
                                           unsigned OnOff );
```

Parameter

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	0: Tx for the port disabled. 1: Tx for the port enabled.

Table 17.13: IP_PHY_MICREL_SWITCH_ConfigTxEnable() parameter list

Chapter 18

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.

18.1 Runtime configuration

Every driver folder includes a configuration file with implementations of runtime configuration functions explained in this chapter. These functions can be customized.

18.1.1 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 119 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, minimum size optimized
 *      #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 bytes for
 *                                                           // Ethernet header (2 bytes
 *                                                           // type, 2 * 6 bytes MAC,
 *                                                           // 2 bytes padding).
 *      IP_ConfTCPSpace(2 * (mtu - 40), 1 * (mtu - 40));      // Define TCP Tx and Rx
 *                                                           // window size.
 *
 *      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 bytes 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.
 *
 *      Microcontroller system, speed optimized or multiple connections
 *      #define ALLOC_SIZE 0x6000                                // 24 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(12, 256);                                 // Small buffers.
 *      IP_AddBuffers(6, mtu + 16);                           // Big buffers. Size should
 *                                                           // be mtu + 16 bytes for
 *                                                           // Ethernet header (2 bytes
 *                                                           // type, 2 * 6 bytes MAC,
 *****/

```

```

*
*      IP_ConfTCPSpace(3 * (mtu - 40), 3 * (mtu - 40)); // 2 bytes padding).
*                                                         // 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 bytes for
*                                  // Ethernet header (2 bytes
*                                  // type, 2 * 6 bytes MAC,
*                                  // 2 bytes padding).
*      IP_ConfTCPSpace(8 * (mtu - 40), 8 * (mtu - 40)); // Define TCP Tx and Rx
*                                                         // window size.
*/
void IP_X_Config(void) {
    int mtu;

    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_DHCP_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(6, mtu + 16); // Big buffers. Size should be
                                // mtu + 16 bytes for Ethernet
                                // header (2 bytes type, 2 * 6
                                // bytes MAC, 2 bytes padding).

    IP_ConfTCPSpace(3 * (mtu - 40), 3 * (mtu - 40)); // 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
    );
}

```

18.1.2 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.

18.1.3 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

18.1.3.1 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.

18.1.3.2 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 - Extremely 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.
IP_SetMTU(0, mtu);                 // to RFC, 1500 is max. for Ethernet
IP_AddBuffers(4, 256);             // Maximum Transmission Unit is 1500
IP_AddBuffers(2, mtu + 16);        // for ethernet by default
                                   // Small buffers.
                                   // 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), 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.
IP_SetMTU(0, mtu);                 // to RFC, 1500 is max. for Ethernet
IP_AddBuffers(8, 256);             // Maximum Transmission Unit is 1500
IP_AddBuffers(4, mtu + 16);        // for ethernet by default
                                   // Small buffers.
                                   // 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(8 * (mtu-40), 8 * (mtu-40)); // Define TCP Tx and Rx window size
```

18.2 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).

18.2.1 Compile-time configuration switches

Type	Symbolic name	Default	Description
System configuration macros			
N	<code>IP_IS_BIGENDIAN</code>	--	Macro to define if a big endian target is used.
Statistics configuration macros			
N	<code>IP_SUPPORT_STATS</code>	0	Macro used as default value for all <code>IP_SUPPORT_STATS_*</code> defines. Leave this to 0 if you want to enable only specific stats defines.
N	<code>IP_SUPPORT_STATS_IFACE</code>	<code>IP_SUPPORT_STATS</code>	Macro to define if the embOS/IP interface statistics should be available.
Debug macros			
N	<code>IP_DEBUG</code>	0	Macro to define the debug level of the embOS/IP build. Refer to <i>Debug level</i> on page 427 for a description of the different debug level.
N	<code>IP_SUPPORT_PROFILE</code>	0	Macro to define if the embOS/IP API profiling support for System-View is used. For more information regarding SystemView please refer to https://www.segger.com/system-view.html .
N	<code>IP_SUPPORT_PROFILE_END_CALL</code>	0	Macro to define if the embOS/IP API profiling support for System-View recognizes the exact end of functions as well. For more information regarding SystemView please refer to https://www.segger.com/system-view.html .
Optimization macros			

Type	Symbolic name	Default	Description
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.
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.

18.2.2 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 795). 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 19

Internet Protocol version 6 (IPv6) (Add-on)

The embOS/IP implementation of the Internet Protocol version 6 (IPv6) allows you a fast and easy transition from IPv4 only applications to dual IPv4 and IPv6 applications.

19.1 embOS/IP IPv6

The embOS/IP IPv6 add-on 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 following table shows the contents of the embOS/IP IPv6 add-on root directory:

Directory	Content
Application\	Contains the example application to test the IPv6 implementation.
Config\	Contains the embOS/IP IPv6 related configuration files.
Inc\	Contains the required include files.
IP\	Contains the IPv6 sources: IPV6_DNSC.c IPV6_ICMPv6.c IPV6_ICMPv6_MLD.c IPV6_ICMPv6_NDP.c IPV6_Int.h IPV6_IPv6.c IPV6_IPv6.h IPV6_TCP.c IPV6_TCP_Rx.c IPV6_TCP_Sock.c IPV6_TCP_Tx.c IPV6_UDP.c IPV6_UDP_Sock.c

Supplied directory structure of embOS/IP IPv6 add-on package

19.2 Feature list

- Low memory footprint
- Easy to implement
- Internet Protocol version 6 (IPv6)
- Internet Control Message Protocol (ICMPv6)
- Neighbor Discovery Protocol (NDP)
- Multicast Listener Discovery (MLD)
- Stateless Address autoconfiguration (SLAAC)
- Standard socket interface
- No configuration required

19.3 IPv6 backgrounds

IPv6 is a network layer protocol. It is the most recent version of the Internet Protocol and is intended to replace IPv4 in the near future. The name IPv6 is commonly used generic term for an internet protocol suite and summarizes the following protocols:

- Internet Protocol version 6 (IPv6)
- Internet Control Message Protocol (ICMPv6)
- Neighbor Discovery Protocol (NDP)
- Multicast Listener Discovery (MLD)

The IPv6 has a larger address space, supports stateless address autoconfiguration and makes extensive use of multicasting. The IPv6 protocol header is designed to simplify processing by routers and is extensible for new requirements.

Application layer	DHCPv6, DNSv6, ...
Transport layer	TCP, UDP
Network layer	IPv6, ICMPv6 (MLD, NDP)
Link layer	Ethernet (IEEE 802.3), ...

The IPv6 header has a fixed length of 40 bytes and does not include any option. Contrary to IPv4, options are stored in extension headers. The benefit of this separation is that a router never needs to parse the header so that the processing is more efficient although the header size is at least twice the size of an IPv4 header.

The most conspicuous difference between IPv4 and IPv6 is the length of the address. The length of an IPv6 address is 128 bits, compared to 32 bits in IPv4. The address space therefore has 2^{128} addresses. Today public IPv4 addresses have become relatively scarce. This problem can be solved by using IPv6.

Internet Protocol header comparison

The IPv6 header contains the Internet Protocol version, traffic classification options the length of the payload, the optional extension or payload which follows the header, a hop limit and the source and destination addresses.

IPv4 header

Version	IHL	TOS	Total length	
Identification			Flags	Fragment offset
Time to live	Protocol		Header checksum	
Source address				
Destination address				
Options				Padding

Legend

	New field in IPv6
	Name and position changed in IPv6
	Field not kept in IPv6
	Field's name kept from IPv4 to IPv6

IPv6 header

Version	Traffic class	Flow label		
Payload length		Next header	Hop limit	
Source address				
Destination address				

Compared to the IPv4 header, the number of header elements is simplified. Unnecessary or ambiguous elements such as header checksum or IHL removed. Other elements like Time to live, which is in practice used as hop limit, are renamed.

19.3.1 IPv6 address types

There are three types of IPv6 addresses:

- Unicast
- Multicast
- Anycast

IPv6 addresses are represented as eight groups of four hexadecimal digits separated by colons.

For example: `fe80:0000:0000:0000:0222:c7ff:feff:ff23` is a link local unicast address.

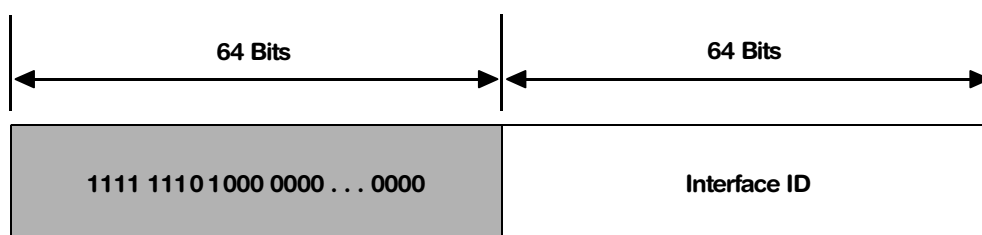
IPv6 addresses may be abbreviated to shorter notations. The following rules for abbreviation are defined by RFC 5952 "A Recommendation for IPv6 Address Text Representation".

- One or more leading zeroes from any groups of hexadecimal digits are removed; this is usually done to either all or none of the leading zeroes. For example, the group 0222 is converted to 222.
- Consecutive sections of zeroes are replaced with a double colon (::). The double colon can only be used once in an address, as multiple use would render the address indeterminate.

Using the recommended rules for abbreviation the textual representation of the example IPv6 address can be simplified to `fe80::222:c7ff:feff:ff23`.

19.3.1.1 Link-local unicast addresses

An IPv6 link-local unicast address is always automatically configured for each interface. It is required for Neighbor Discovery and DHCPv6 processes. A link-local address is also useful in single-link networks with no router. It can be used to communicate between hosts on a single-link. IPv6 link-local addresses will never be forwarded by an IPv6 router.

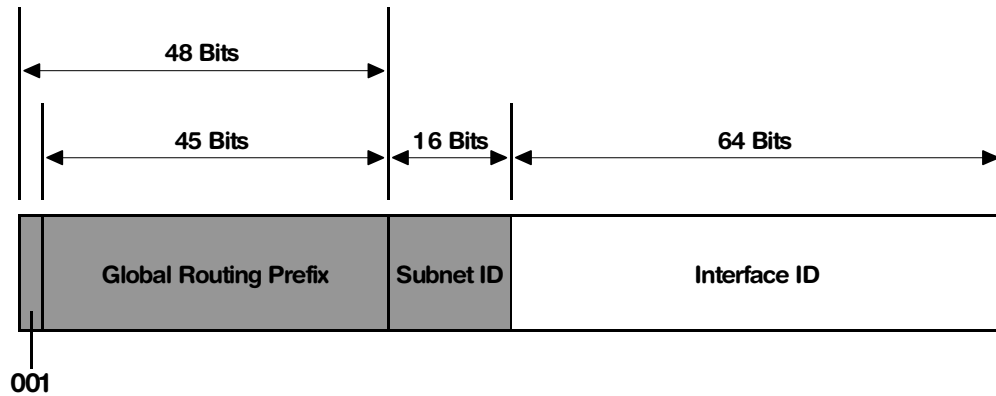


The first 64 bits are link-local address prefix. Prefixes for IPv6 subnets are expressed in the same way as Classless Inter-Domain Routing (CIDR) notation for IPv4. An IPv6 prefix is written in address/prefix-length notation. The prefix for link-local addresses is always `fe80::/64`.

The second 64 bits are the interface identifier. The interface identifier is a modified EUI-64 identifier. Please refer to the appendix of RFC 4291 "IP Version 6 Addressing Architecture" for further information.

19.3.1.2 Global unicast addresses

IPv6 global unicast addresses are the counterpart to IPv4 public addresses. They are globally routable and reachable on the IPv6 Internet.



RFC 3587 defines global addresses that are currently being used on the IPv6 Internet. According to RFC 3587 an IPv6 global unicast address consists of four parts:

- **Fixed high-order bits** - 3 bits: 001
- **Global routing prefix** - 45 bits: Together with the three high-order bits builds the global routing prefix the site prefix. A site is an autonomously operating network that is connected to the IPv6 Internet. A site prefix identifies an individual site of an organization, so that routers can forward IPv6 traffic matching the 48-bit prefix to the routers of the organization's site.
- **Subnet ID** - 16 bits: Used to identify subnets within an organization's site.
- **Interface ID** - 64 bits: Normally, the interface identifier is a modified EUI-64 identifier.

19.3.2 Further reading

This chapter explains the usage of the embOS/IP IPv6 add-on. It describes all functions which are required to build a network application using IPv6. 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 embOS/IP IPv6 add-on. 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.

19.3.2.1 Request for Comments (RFC)

RFC#	Description
[RFC 2460]	Internet Protocol, Version 6 (IPv6) Specification Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc2460.txt
[RFC 2464]	Transmission of IPv6 Packets over Ethernet Networks Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc2464.txt
[RFC 2710]	Multicast Listener Discovery (MLD) for IPv6 Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc2710.txt
[RFC 3306]	Unicast-Prefix-based IPv6 Multicast Addresses Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc3306.txt
[RFC 3587]	IPv6 Global Unicast Address Format Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc3587.txt
[RFC 3590]	Source Address Selection for the Multicast Listener Discovery (MLD) Protocol Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc3590.txt
[RFC 3810]	Multicast Listener Discovery Version 2 (MLDv2) for IPv6 Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc3810.txt
[RFC 4291]	IP Version 6 Addressing Architecture Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc4291.txt
[RFC 4443]	Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc4443.txt
[RFC 4861]	Neighbor Discovery for IP version 6 (IPv6) Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc4861.txt
[RFC 4862]	IPv6 Stateless Address Autoconfiguration Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc4862.txt
[RFC 5952]	A Recommendation for IPv6 Address Text Representation Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc5952.txt

19.3.2.2 Related books

- [Hagen] - IPv6 Essentials, Silvia Hagen
ISBN: 978-1449319212
- [Davies] - Understanding IPv6, Joseph Davies
ISBN: 978-0735659148

19.4 Include IPv6 to your embOS/IP start project

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

- Step 1: Open an embOS/IP project and compile it.
- Step 2: Add embOS/IP IPv6 add-on to the start project.
- Step 3: Build the project and test it.

The following steps presume that you use a project with the recommended project structure. If your project structure differs, keep in mind that you potentially have to add additional directories to your include path.

19.4.1 Open an embOS/IP project and compile it

To add the embOS/IP IPv6 add-on to your project, you need a running embOS/IP project. For a step by step tutorial to setup an embOS/IP project refer to *Running embOS/IP on target hardware* on page 35.

19.4.2 Add the embOS/IP IPv6 add-on to the start project

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

- IP

The embOS/IP IPv6 add-on default configuration is preconfigured with valid values, which matches the requirements of the most applications.

19.4.2.1 Enable IPv6 support

To enable the IPv6 support, you have to add the following define to your `IP_Conf.h`:

```
#define IP_SUPPORT_IPV6      1
```

Build the project. It should compile without error and warnings.

To include IPv6 in your application you need to add the IPv6 related protocols by calling `IP_IPv6_Add()`.

Add the function call to `IP_X_Config()` as shown below:

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

    IP_AssignMemory(_aDrvPool, sizeof(_aDrvPool)); // Assigning memory should
                                                    // be the first thing
    IP_AddEtherInterface(&IP_Driver_STM32F207);    // Add ethernet driver for your
                                                    // hardware
    IP_SetHWAddr("\x00\x22\xC7\xFF\xFF\x23");      // MAC addr: Needs to be
                                                    // unique for production units
    IP_DHCP_Activate(0, "TARGET", NULL, NULL);      // Request an IPv4 address
    //
    // Run-time configure buffers.
    // The default setup will do for most cases.
    //
    Mtu = 1500;                                     // 576 is minimum for IPv4,
                                                    // 1280 is minimum for IPv6,
                                                    // 1500 is max. for Ethernet

    IP_SetMTU(0, Mtu);
    IP_AddBuffers(12, 256);                         // Small buffers.
    IP_AddBuffers(6, Mtu + 16);                     // Big buffers.
    IP_ConfTCPSpace(3 * (Mtu - 60), 3 * (Mtu - 60)); // Define TCP Tx and Rx window size
    //
    // Define log and warn filter
    //
    IP_SetWarnFilter(0xFFFFFFFF);                   // 0xFFFFFFFF: Do not filter:
                                                    // 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
        | IP_MTYPE_IPV6                 // Output IPv6 status messages
    );

    //
    // Add protocols to the stack
    //
    IP_UDP_Add();           // Add transport protocol: UDP.
    IP_TCP_Add();           // Add transport protocol: TCP.
    IP_ICMP_Add();          // Add ICMPv4.
    IP_IPV6_Add(0);         // Add IPv6, includes ICMPv6, MLD and NDP.
}

```

The source code excerpt is used by a target which requests an IPv4 address from a DHCP server. The link-local IPv6 address will be generated automatically during initialization. Please ensure that the MAC address of your target is unique in your network segment, since it is used to build the interface identifier part of the IPv6 address.

19.4.2.2 Configure the MTU and the Tx/Rx window sizes

The Maximum Transmission Unit (MTU) is the largest number of payload bytes that can be sent in a packet. 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.

As opposed to IPv4, which requires at least 576 bytes as MTU, RFC2460 defines that IPv6 has to use at least 1280 bytes. If you do not use the Ethernet maximum of 1500 bytes, check in your `IP_X_Config()` that the MTU size is not smaller as 1280 bytes.

The TCP transmit and receive window sizes, configured with `IP_ConfTCPSpace()`, should also be checked. An IPv4 header without options is 20 bytes. Together with the TCP header the payload of a normal TCP/IPv4 packet can be up to 1460 bytes.

It is a good approach to calculate the sizes of the transmit window and receive window with the following formula: $x * (MTU - (IP\ header\ size + TCP\ header\ size))$

x is the number of big packets, which are available for each TCP connection.

embOS/IP sample configurations up to embOS/IP version 2.20 always include a call of `IP_ConfTCPSpace()` and computes a matching window sizes for IPv4 targets with this formula.

Example

```
IP_ConfTCPSpace(3 * (Mtu - 40), 3 * (Mtu - 40)); // Define TCP Tx and Rx window size
```

Since the IPv6 header is 40 bytes, the payload of a normal TCP/IPv6 packet is limited to a maximum of 1440 bytes (Max. Ethernet MTU - (IPv6 header size + TCP header size), $1500 - (40 + 20)$). You need to change the transmit window and receive window sizes to ensure the best possible TCP performance.

Example

```
IP_ConfTCPSpace(3 * (Mtu - 60), 3 * (Mtu - 60)); // Define TCP Tx and Rx window size
```

19.4.2.3 Enable terminal output for IPv6 messages

In debug builds of embOS/IP, logging messages can be used. `IP_SetLogFilter()` sets a mask that defines which logging messages should be logged. To output IPv6 related logging messages the message type `IP_MTYPE_IPV6` needs to be added.

Example

```

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
    | IP_MTYPE_IPV6                 // Output IPv6 status messages
);

```

19.4.2.4 Select the start application

For testing of your embOS/IP IPv6 add-on integration, start with the code found in the folder Application. Add one of the applications to your project (for example OS_IP_SimpleServer_IPv6.c)

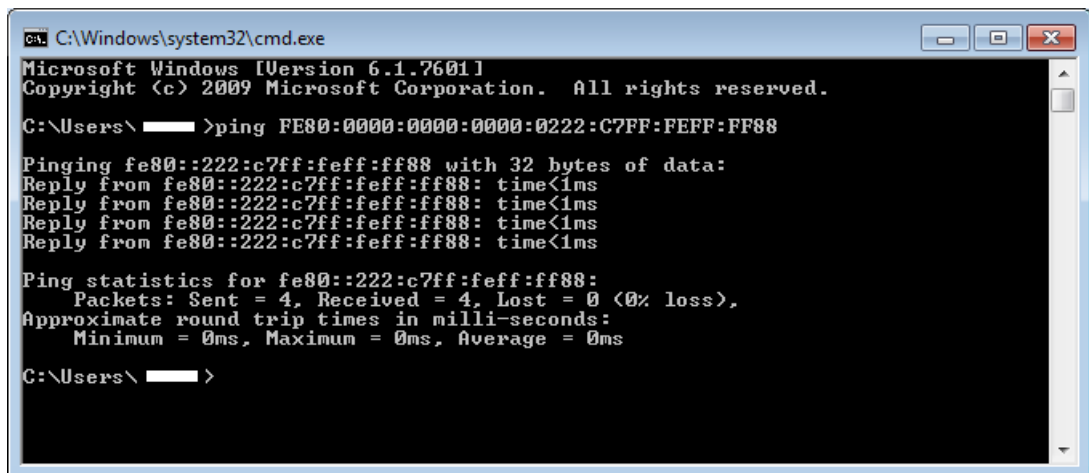
19.4.3 Build the project and test it

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.

A target which uses IPv4 and IPv6 should output similar logging messages as shown below:

```
0:000 MainTask - INIT: Init started.
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
3:000 IP_Task - LINK: Link state changed: Full duplex, 100MHz
3:400 IP_Task - NDP: Link-local IPv6 addr.: FE80:0000:0000:0222:C7FF:FEFF:FF88
added to IFace: 0
4:000 IP_Task - DHCPc: Sending discover!
5:002 IP_Task - DHCPc: IFace 0: Offer: IP: 192.168.1.12, Mask: 255.255.255.0, GW:
192.168.1.1.
6:000 IP_Task - DHCPc: IP addr. checked, no conflicts
6:000 IP_Task - DHCPc: Sending Request.
6:001 IP_Task - DHCPc: IFace 0: Using IP: 192.168.1.12, Mask: 255.255.255.0, GW:
192.168.1.1.
```

ICMPv6 is always activated. This means that you can 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.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\ >ping FE80:0000:0000:0222:C7FF:FEFF:FF88

Pinging fe80::222:c7ff:feff:ff88 with 32 bytes of data:
Reply from fe80::222:c7ff:feff:ff88: time<1ms
Reply from fe80::222:c7ff:feff:ff88: time<1ms
Reply from fe80::222:c7ff:feff:ff88: time<1ms
Reply from fe80::222:c7ff:feff:ff88: time<1ms

Ping statistics for fe80::222:c7ff:feff:ff88:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\Users\ >
```

19.5 Configuration

The embOS/IP IPv6 add-on 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.

19.5.1 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 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).

19.5.2 Compile time configuration switches

Type	Symbolic name	Default	Description
B	IP_SUPPORT_IPV6	0	Enables support for IPv6. Refer to <i>IP_IPV6_Add()</i> on page 442 for further information about enabling IPv6.
N	IP_NDP_MAX_ENTRIES	5	Maximum number of stored NDP entries.
N	IP_IPV6_DNS_MAX_IPV6_SERVER	1	Maximum number of available IPv6 DNS servers.

19.5.3 Runtime configuration

Please refer to *IP_IPV6_Add()* on page 442 for detailed information about runtime configuration.

19.6 API functions

Function	Description
Configuration functions	
<code>IP_IPV6_Add()</code>	Adds IPv6 to the stack.
<code>IP_IPV6_AddUnicastAddr()</code>	Adds an additional unicast address to the stack.
<code>IP_IPV6_ParseIPv6Addr()</code>	Parses an IPv6 address.
<code>IP_IPV6_SetDefHopLimit()</code>	Sets the default hop limit.
<code>IP_ICMPV6_MLD_AddMulticastAddr()</code>	Adds a multicast address.
<code>IP_ICMPV6_MLD_RemoveMulticastAddr()</code>	Removes a multicast address.

Table 19.1: embOS/IP IPv6 API function overview

19.6.1 IP_IPV6_Add()

Description

Adds Internet Protocol version 6 to the stack.

Prototype

```
void IP_IPV6_Add ( unsigned IFaceId );
```

Parameter

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

Table 19.2: IP_IPV6_Add() parameter list

Additional information

The call of `IP_IPV6_Add()` adds and initializes all required IPv6 protocols to the stack. This means that Internet Control Message Protocol version 6 (ICMPv6), Multicast Listener Discovery (MLD) and Neighbor Discovery Protocol (NDP) are added and initialized.

Part of the initialization is the generation of a link-local address, since all IPv6 hosts require such an address. The link-local address is derived from the MAC address of an interface and the link-local prefix FE80::/64. The uniqueness of the address on the subnet is tested using the Duplicate Address Detection (DAD) method.

Note: You need to set the compile time switch `IP_SUPPORT_IPV6` to 1 to enable the stack to work in dual stack mode supporting IPv4 and IPv6.

Example

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

    IP_AssignMemory(_aDrvPool, sizeof(_aDrvPool)); // Assigning memory should
                                                    // be the first thing
    IP_AddEtherInterface(&IP_Driver_STM32F207);    // Add ethernet driver for your
                                                    // hardware
    IP_SetHWAddr("\x00\x22\xC7\xFF\xFF\x23");      // MAC addr: Needs to be
                                                    // unique for production units
    IP_DHCPC_Activate(0, "TARGET", NULL, NULL);    // Request an IPv4 address
    //
    // Run-time configure buffers.
    // The default setup will do for most cases.
    //
    Mtu = 1500;                                     // 576 is minimum for IPv4,
                                                    // 1280 is minimum for IPv6,
                                                    // 1500 is max. for Ethernet

    IP_SetMTU(0, Mtu);
    IP_AddBuffers(12, 256);                         // Small buffers.
    IP_AddBuffers(6, Mtu + 16);                     // Big buffers.
    IP_ConfTCPSpace(3 * (Mtu - 60), 3 * (Mtu - 60)); // Define TCP Tx and Rx window size
    //
    // Define log and warn filter
    //
    IP_SetWarnFilter(0xFFFFFFFF);                   // 0xFFFFFFFF: Do not filter:
                                                    // 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
                    | IP_MTYPE_IPV6               // Output all IPv6 status messages
                    );
    //
    // Add protocols to the stack
    //
```

```
IP_UDP_Add();    // Add transport protocol: UDP.  
IP_TCP_Add();    // Add transport protocol: TCP.  
IP_ICMP_Add();   // Add ICMPv4.  
IP_IPv6_Add();   // Add IPv6, includes ICMPv6, MLD and NDP.  
}
```

19.6.2 IP_IPV6_AddUnicastAddr()

Description

Adds an additional IPv6 address to the given interface.

Prototype

```
int IP_IPV6_AddUnicastAddr(      U8  IFaceId,  
                                const U8 *pAddr );
```

Parameter

Parameter	Description
<code>IFaceId</code>	[IN] Zero-based index of available network interfaces.
<code>pAddr</code>	[IN] Pointer to the 16 byte IPv6 address which should be added to the network interface.

Table 19.3: IP_IPV6_AddUnicastAddr() parameter list

Return value

0 - OK. IPv6 address added to the network interface.

1 - Error. IPv6 address could not be added to the network interface.

Additional notes

A link-local address (prefix FE80::/64) derived from the MAC address of the interface has been set during initialization of the stack.

19.6.3 IP_IPV6_ParseIPv6Addr()

Description

Checks if an IPv6 address string in colon hexadecimal notation is valid and transforms it into a byte stream (big endian byte order).

Prototype

```
int IP_IPV6_ParseIPv6Addr (          IPV6_ADDR  *pIPv6Addr,
                                const char      *sHost );
```

Parameter

Parameter	Description
pIPv6Addr	[OUT] Pointer to an IPv6 address structure to store the byte stream.
sHost	[IN] Pointer to the IPv6 address string to parse.

Table 19.4: IP_IPV6_AddUnicastAddr() parameter list

Return value

- 0: OK.
- 1: Error. Not every character in address are hexadecimal values (0-f) or colons (:).
- 2: Error. Too much characters for 16bit block.
- 3: Error. Illegal number of colons in a row (":::").
- 4: Error. "::" is used twice.
- 5: Error. Address string too long.
- 6: Error. Too much colons.
- 7: Error. Parameter invalid

Additional information

IPv6 addresses are represented in eight 16-bit blocks. Each 16-bit block is converted to a 4-digit hexadecimal number and separated by colons.

(For example: 2001:0db8:0000:0000:0001:0000:0234:0001)

The representation can be simplified by suppressing the leading zeros within each 16-bit block.

(For example: 2001:db8:0:0:1:0:234:1)

IPv6 addresses that contain long sequences of zeros can be further simplified. A single contiguous sequence of 16-bit blocks set to 0 in the colon hexadecimal format can be compressed to '::'.

(For example: 2001:db8::1:0:234:1).

IP_IPV6_ParseIPv6Addr() checks if an IPv6 address string is valid and

Example

```
static void _ParseAndPrintIPv6Addr (void) {
    IPV6_ADDR IPv6Addr;

    IP_IPV6_ParseIPv6Addr(&IPv6Addr, "2001:db8::1:0:234:1");
    IP_Logf_Application("IPv6 addr.: %n", IPv6Addr.Union.aU8);
}
```

Output:

```
IPv6 addr.: 2001:0DB8:0000:0000:0001:0000:0234:0001
```

19.6.4 IP_ICMPV6_MLD_AddMulticastAddr()

Description

Adds an additional multicast address to the given interface.

Prototype

```
int IP_ICMPV6_MLD_AddMulticastAddr(          U8  IFaceId,
                                           const U8 *pMultiCAddr );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.
pMultiCAddr	[IN] Pointer to the 16 byte IPv6 multicast address which should be added to the network interface.

Table 19.5: IP_ICMPV6_MLD_AddMulticastAddr() parameter list

Return value

- 0 - OK. IPv6 multicast address added to the network interface.
- 1 - Error. IPv6 multicast address could not be added to the network interface.

Additional notes

The IPv6 multicast addresses All-Routers (FF01:0:0:0:0:0:0:2) and All-Nodes (FF01:0:0:0:0:0:0:1) are always automatically added to the network interface, since they are required for correct function of the IPv6 implementation.

19.6.5 IP_ICMPV6_MLD_RemoveMulticastAddr()

Description

Removes a multicast address from the given interface.

Prototype

```
int IP_ICMPV6_MLD_RemoveMulticastAddr(      U8  IFaceId,
                                             const U8 *pMultiCAddr );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.
pMultiCAddr	[IN] Pointer to the 16 byte IPv6 multicast address which should be removed.

Table 19.6: IP_ICMPV6_MLD_RemoveMulticastAddr() parameter list

Return value

0 - OK. IPv6 multicast address removed from the network interface.
 1 - Error. IPv6 multicast address could not be removed.

19.6.6 IP_IPV6_SetDefHopLimit()

Description

Sets a default hop limit.

Prototype

```
int  IP_IPV6_SetDefHopLimit ( U8 IFaceId,  
                               U8 HopLimit );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.
HopLimit	[IN] Default hop limit.

Table 19.7: IP_IPV6_SetDefHopLimit() parameter list

Return value

0 - OK. Hop limit set.

1 - Error. Hop limit could not be set.

19.7 Socket API extensions

The socket interface was developed for Unix in the early 1980s and has also been implemented on a wide variety of non-Unix systems. Today it is the de facto standard Application Program Interface (API) for TCP/IP applications.

With the new version of the internet protocol some changes were required to support IPv6. RFC 3493 "*Basic Socket Interface Extensions for IPv6*" describes the recommended extensions to the socket API.

In the current version of the embOS/IP IPv6 add-on are the following extensions included.

Structures

Structures	Description
sockaddr_in6	Structure to handle IPv6 addresses.

Table 19.8: embOS/IP socket API extensions overview

Socket options

Socket options	Description
IPV6_JOIN_GROUP	Join an IPv6 multicast group on a specified local interface.
IPV6_LEAVE_GROUP	Leave an IPv6 multicast group on a specified local interface.

Table 19.9: embOS/IP socket options overview

19.7.1 Structure sockaddr_in6

Description

Structure to handle IPv6 addresses.

Prototype

```
typedef struct sockaddr_in6 {  
    U16      sin6_family;  
    U16      sin6_port;  
    U32      sin6_flowinfo;  
    IPV6_ADDR sin6_addr;  
    U32      sin6_scope_id;  
} SOCKADDR_IN6;
```

Member	Description
sin6_family	Protocol family (AF_INET6).
sin6_port	Transport layer port stored in network byte order.
sin6_flowinfo	Flow information.
sin6_addr	16-bytes IPv6 address.
sin6_scope_id	Set of interfaces for a scope.

Table 19.10: Structure sockaddr_in6 member list

This structure is required to pass IPv6 addresses to socket interface functions like `accept()`, `bind()`, `connect()`, `recvfrom()` and `sendto()`. For further information about usage, please refer to *Porting an IPv4 application to IPv6* on page 451 and *API functions* on page 208 for details about the usage of the socket API functions.

19.8 Porting an IPv4 application to IPv6

TCP/IP applications written using the socket API have in the past enjoyed a high degree of portability. This portability was kept in mind while the socket API was extended to support IPv6. Complete compatibility for existing IPv4 applications is always ensured.

Besides smaller enhancements like some new socket options, a new socket address structure has been added to carry IPv6 addresses.

The following sections demonstrate, using the supplied IPv4 example applications, which parts have to be changed to communicate via IPv6. All examples are also part of the embOS/IP IPv6 add-on shipment.

19.8.1 Porting an IPv4 server application to IPv6

The main difference between an IPv4 and an IPv6 socket application lay in the functions which pass a socket address structure as a parameter. The relevant functions are `accept()`, `bind()`, `connect()`, `recvfrom()` and `sendto()`.

The prototype of the `sockaddr_in6` structure is shown below.

```
typedef struct sockaddr_in6 {
    U16      sin6_family;    // AF_INET6
    U16      sin6_port;      // Transport layer port stored in network byte order.
    U32      sin6_flowinfo;  // IPv6 flow information
    IPV6_ADDR sin6_addr;     // IPv6 address
    U32      sin6_scope_id;  // Set of interfaces for a scope
} SOCKADDR_IN6;
```

The `sockaddr_in6` structure is 28 bytes. For further information, please refer to *Structure sockaddr_in6* on page 450.

embOS/IP IPv6 add.on comes with three version of `OS_IP_SimpleServer`.

File	Description
<code>OS_IP_SimpleServer.c</code>	IPv4 version of the simple TCP server example. Server listens on port 23 for IPv4 clients.
<code>OS_IP_SimpleServer_IPv6.c</code>	IPv6 version of the simple TCP server example. Server listens on port 23 for IPv6 clients.
<code>OS_IP_SimpleServer_IPv4_IPv6.c</code>	Dual stack version of the simple TCP server example. Server listens on port 23 for IPv4 and IPv6 clients.

Table 19.11: embOS/IP IPv6 add-on example applications

19.8.1.1 TCP/IPv4 server sample code

The supplied example `OS_IP_SimpleServer.c` is a simple Telnet server listening on port 23 that outputs the current system time for each character received. It uses `bind()` to assign a socket address to a socket and `accept()` to create a new connected socket. To assign a socket address to a socket a `sockaddr` structure needs to be initialized and used as parameter for `bind()`.

The following excerpt of `IP_OS_SimpleServer.c` shows a code snippet, which creates an IPv4 socket, binds it to TCP port 23 and sets it into listening state.

```

/*****
*
*      _ListenAtTcpAddr()
*
* Function description
*   Creates a socket for port 23 and sets it into listening state.
*   The only step left is to call accept() to actually wait for a
*   a client to connect.
*
* Return value
*   O.K. : Socket handle.
*   Error: SOCKET_ERROR.
*/
static int _ListenAtTcpAddr(void) {
    struct sockaddr_in Addr;
    int             hSock;
    int             r;

    hSock = socket(AF_INET, SOCK_STREAM, 0);
    if (hSock != SOCKET_ERROR) {
        IP_MEMSET(&Addr, 0, sizeof(Addr));
        Addr.sin_family    = AF_INET;
        Addr.sin_port      = htons(23);
        Addr.sin_addr.s_addr = INADDR_ANY;
        r = bind(hSock, (struct sockaddr*)&Addr, sizeof(Addr));
        if (r != 0) {
            hSock = SOCKET_ERROR;
        } else {
            r = listen(hSock, 1);
            if (r != 0) {
                hSock = SOCKET_ERROR;
            }
        }
    }
    return hSock;
}

```

The socket creation is done with the following line of code:

```
hSock = socket(AF_INET, SOCK_STREAM, 0);
```

`AF_INET` is the address family for IPv4. The rest of the code snippet fills the `sockaddr_in` structure and pass it, together with the size of the `sockaddr_in` structure, to `bind()`.

The IPv4 socket address structures `sockaddr_in` and `sockaddr` have a size of 16 bytes. For further information about the `sockaddr_in` structure, please refer to the *Structure sockaddr_in* on page 239.

The following excerpt of `IP_OS_SimpleServer.c` shows a code snippet, which accepts connections using the socket returned by the call of `_ListenAtTcpAddr()`.

```

/*****
*
*      _TelnetTask()
*
* Function description
*   Creates a parent socket and handles clients that connect to the
*   server. This sample can handle one client at the same time. Each
*   client that connects creates a child socket that is then passed
*   to the process routine to detach client handling from the server
*   functionality.
*/
static void _TelnetTask(void) {
    struct sockaddr Addr;
    int          AddrLen;
    int          hSockParent;
    int          hSockChild;

    AddrLen = sizeof(Addr);
    while (1) {
        //
        // Try until we get a valid parent socket.
        //
        hSockParent = _ListenAtTcpAddr();
        if (hSockParent == SOCKET_ERROR) {
            OS_Delay(5000);
            continue; // Error, try again.
        }
        while (1) {
            //
            // Try until we get a valid child socket.
            // Typically accept() will only return when
            // a valid client has connected.
            //
            hSockChild = accept(hSockParent, &Addr, &AddrLen);
            if (hSockChild == SOCKET_ERROR) {
                continue; // Error, try again.
            }
            IP_Logf_Application("New client (%i) accepted.", Addr.sin_addr.s_addr);
            _Process(hSockChild); // Process the client.
            closesocket(hSockChild); // Close connection to client from our side (too).
        }
    }
}

```

`accept()` returns a new connected socket which is used to transfer data between the embOS/IP host and the client. The optional output parameters `pAddr` and `pAddrLen` of `accept()` are still only used for debugging purposes. We output the IPv4 address of the client after connecting to our host. The output should be similar to the following:

```
Telnet - New client (192.168.11.29) accepted.
```

The new connected socket is passed to the function `_Process()` which handles the data transmission. When the process returns, the socket will be closed and the host can process further client requests.

For further information about `accept()`, please refer to *accept()* on page 209.

Required changes to port the TCP/IPv4 server sample code to TCP/IPv6

To port these simple telnet style server to IPv6, `_ListenAtTcpAddr()` and `_TelnetTask()` has to be modified. The rest of the example `IP_OS_SimpleServer.c` keeps untouched.

`_ListenAtTcpAddr()` needs to create an IPv6 socket instead of an IPv4 socket and the `sockaddr_in` structure has to be replaced by a `sockaddr_in6` structure.

The socket creation changes from `hSock = socket(AF_INET6, SOCK_STREAM, 0);` to `hSock = socket(AF_INET6, SOCK_STREAM, 0);`

`AF_INET6` is the address family for IPv6.

The modified function `_ListenAtTcpAddr()` looks like the code snippet below.

```

/*****
*
*      _ListenAtTcpAddr()
*
* Function description
*   Creates a socket for port SERVER_PORT and sets it into listening
*   state. The only step left is to call accept() to actually wait for
*   a client to connect.
*
* Return value
*   O.K. : Socket handle.
*   Error: SOCKET_ERROR .
*/
static int _ListenAtTcpAddr(void) {
    struct sockaddr_in6 Addr;
    int      hSock;
    int      r;

    hSock = socket(AF_INET6, SOCK_STREAM, 0);
    if (hSock != SOCKET_ERROR) {
        Addr.sin6_family   = AF_INET6;
        Addr.sin6_port     = htons(23);
        Addr.sin6_flowinfo = 0;
        IP_MEMSET(&Addr.sin6_addr.Union.aU8[0], 0, IPV6_ADDR_LEN);
        Addr.sin6_scope_id = 0;
        r = bind(hSock, (struct sockaddr*)&Addr, sizeof(Addr));
        if (r != 0) {
            hSock = SOCKET_ERROR;
        } else {
            r = listen(hSock, 1);
            if (r != 0) {
                hSock = SOCKET_ERROR;
            }
        }
    }
    return hSock;
}

```

Compared to the IPv4 version of these function, `AF_INET6` is used to specify the address family to create an IPv6 socket. The port number is still 23 and the address element `sin6_addr` is set to zero, which means that the socket will be bound to all available interfaces. The new elements, `sin6_flowinfo` and `sin6_scope`, are set to zero.

The function `_TelnetTask()` is nearly untouched. The only change is the `sockaddr_in6` structure instead of the `sockaddr` structure used in the IPv4 code.

```

/*****
*
*      _TelnetTask()
*
* Function description
*   Creates a parent socket and handles clients that connect to the
*   server. This sample can handle one client at the same time. Each
*   client that connects creates a child socket that is then passed
*   to the process routine to detach client handling from the server
*   functionality.
*/
static void _TelnetTask(void) {
    struct sockaddr_in6 Addr;
    int AddrLen;
    int hSockParent;
    int hSockChild;

    AddrLen = sizeof(Addr);
    while (1) {
        //
        // Try until we get a valid parent socket.
        //
        hSockParent = _ListenAtTcpAddr();
        if (hSockParent == SOCKET_ERROR) {
            OS_Delay(5000);
            continue; // Error, try again.
        }
        while (1) {
            //
            // Try until we get a valid child socket.
            // Typically accept() will only return when
            // a valid client has connected.
            //
            hSockChild = accept(hSockParent, (struct sockaddr*)&Addr, &AddrLen);
            if (hSockChild == SOCKET_ERROR) {
                continue; // Error, try again.
            }
            IP_Logf_Application("New client (%n) accepted.", Addr.sin6_addr.Union.aU8);
            _Process(hSockChild); // Process the client.
            closesocket(hSockChild); // Close connection to client from our side (too).
        }
    }
}

```

The optional output parameters `pAddr` and `pAddrLen` of `accept()` are still only used for debugging purposes. We output the IPv6 address of the client after connecting to our host. The output should be similar to the following:

```
Telnet - New client (FE80:0000:0000:0000:76D4:35FF:FE8B:5BE5) accepted.
```

The supplied example `OS_IP_SimpleServer_IPv6.c` includes all the mentioned changes. You should start with this example to comprehend the code changes.

19.8.1.2 Dual stack TCP server sample code

The embOS/IP IPv6 add-on provides IPv4 and IPv6 protocol stacks in the same network node. This means that embOS/IP together with the IPv6 add-on is the ideal starting point to implement applications which can facilitate native communications between nodes using either IPv4 or IPv6 or both.

In the transition phase from IPv4 to IPv6 most server applications need to accept connections from IPv4 clients and from IPv6 clients. The supplied example application `OS_IP_SimpleServer_IPv4_IPv6.c` demonstrates a possible way to implement such a TCP server application.

The supplied example `OS_IP_SimpleServer_IPv4_IPv6.c` is a simple Telnet server listening on port 23 that outputs the current system time for each character received.

The following excerpt of `IP_OS_SimpleServer_IPv4_IPv6.c` shows a code snippet, which creates an IPv4 socket and an IPv6 socket, binds both to TCP port 23 and sets both into listening state. To enhance readability of the example code socket creation and binding are implemented as functions.

```

/*****
*
*      _ListenAtTcpPort()
*
* Function description
*   Creates a socket, binds it to a port and sets the socket into
*   listening state.
*
* Parameter
*   IPProtVer - Protocol family which should be used (PF_INET or PF_INET6).
*   Port      - Port which should be to wait for connections.
*
* Return value
*   O.K. : Socket handle.
*   Error: SOCKET_ERROR.
*/
static int _ListenAtTcpPort(unsigned IPProtVer, U16 Port) {
    int hSock;
    int r;

    //
    // Create socket
    //
    hSock = _CreateSocket(IPProtVer);
    if (hSock != SOCKET_ERROR) {
        //
        // Bind it to the port
        //
        r = _BindAtTcpPort(IPProtVer, hSock, Port);
        //
        // Start listening on the socket.
        //
        if (r != 0) {
            hSock = SOCKET_ERROR;
        } else {
            r = listen(hSock, 1);
            if (r != 0) {
                hSock = SOCKET_ERROR;
            }
        }
    }
    return hSock;
}

```


The function used to create either an IPv4 or an IPv6 socket is listed below:

```

/*****
*
*      _CreateSocket()
*
* Function description
*   Creates a socket for the requested protocol family.
*
* Parameter
*   IPProtVer - Protocol family which should be used (PF_INET or PF_INET6).
*
* Return value
*   O.K. : Socket handle.
*   Error: SOCKET_ERROR .
*/
static int _CreateSocket(unsigned IPProtVer) {
    int hSock;

    hSock = SOCKET_ERROR;
    //
    // Create IPv6 socket
    //
    if (IPProtVer == PF_INET6) {
        hSock = socket(AF_INET6, SOCK_STREAM, 0);
    }
    //
    // Create IPv4 socket
    //
    if (IPProtVer == PF_INET) {
        hSock = socket(AF_INET, SOCK_STREAM, 0);
    }
    return hSock;
}

```

The function used to bind either an IPv4 or an IPv6 socket is listed below:

```

/*****
*
*      _BindAtTcpPort()
*
* Function description
*   Binds a socket to a port.
*
* Parameter
*   IPProtVer - Protocol family which should be used (PF_INET or PF_INET6).
*   hSock      - Socket handle
*   Port       - Port which should be to wait for connections.
*
* Return value
*   O.K. : Socket handle.
*   Error: SOCKET_ERROR .
*/
static int _BindAtTcpPort(unsigned IPProtVer, int hSock, U16 LPort) {
    int r;

    //
    // Bind it to the port
    //
    if (IPProtVer == PF_INET6) {
        struct sockaddr_in6 Addr;

        IP_MEMSET(&Addr, 0, sizeof(Addr));
        Addr.sin6_family = AF_INET6;
        Addr.sin6_port = htons(LPort);
        Addr.sin6_flowinfo = 0;
        IP_MEMSET(&Addr.sin6_addr, 0, 16);
        Addr.sin6_scope_id = 0;
        r = bind(hSock, (struct sockaddr*)&Addr, sizeof(Addr));
    }
    if (IPProtVer == PF_INET) {
        struct sockaddr_in Addr;

```

```

    IP_MEMSET(&Addr, 0, sizeof(Addr));
    Addr.sin_family      = AF_INET;
    Addr.sin_port        = htons(LPort);
    Addr.sin_addr.s_addr = INADDR_ANY;
    r = bind(hSock, (struct sockaddr*)&Addr, sizeof(Addr));
}
return r;
}

```

To handle client requests on both sockets within one task `select()` is used. For further information to `select()`, please refer to *select()* on page 226.

To accept connections on both sockets the listening IPv4 socket and the listening IPv6 socket are added to the read set. `select()` returns if data is available on one of these sockets and `accept()` is called to handle the new connection.

```

/*****
*
*      _TelnetTask()
*
* Function description
*   Creates a parent socket and handles clients that connect to the
*   server. This sample can handle one client at the same time. Each
*   client that connects creates a child socket that is then passed
*   to the process routine to detach client handling from the server
*   functionality.
*/
static void _TelnetTask(void) {
    IP_fd_set ReadFds;
    int        hSockParent4;
    int        hSockParent6;
    int        hSockChild;
    int        r;

    //
    // Try until we get a valid IPv4 parent socket and a valid IPv6 parent socket.
    //
    while (1) {
        hSockParent4 = _ListenAtTcpPort(PF_INET, SERVER_PORT);
        if (hSockParent4 == SOCKET_ERROR) {
            OS_Delay(2000);
            continue; // Error, try again.
        }
        break;
    }
    while (1) {
        hSockParent6 = _ListenAtTcpPort(PF_INET6, SERVER_PORT);
        if (hSockParent6 == SOCKET_ERROR) {
            closesocket(hSockParent4);
            OS_Delay(2000);
            continue; // Error, try again.
        }
        break;
    }
    //
    // Wait for a connection on one of the both sockets and process the data
    // requests after accepting the connection.
    //
    while (1) {
        IP_FD_ZERO(&ReadFds); // Clear the set
        IP_FD_SET(hSockParent4, &ReadFds); // Add IPv4 socket to the set
        IP_FD_SET(hSockParent6, &ReadFds); // Add IPv6 socket to the set
        r = select(&ReadFds, NULL, NULL, 5000); // Check for activity. Wait 5 seconds
        if (r <= 0) {
            continue;
        }
        //
        // Check if the IPv4 socket is ready
    }
}

```

```

//
if (IP_FD_ISSET(hSockParent4, &ReadFds)) {
    hSockChild = accept(hSockParent4, NULL, NULL);
    if (hSockChild == SOCKET_ERROR) {
        continue;                // Error, try again.
    }
    IP_Logf_Application("New IPv4 client accepted.");
}
//
// Check if the IPv6 socket is ready
//
else if (IP_FD_ISSET(hSockParent6, &ReadFds)) {
    hSockChild = accept(hSockParent6, NULL, NULL);
    if (hSockChild == SOCKET_ERROR) {
        continue;                // Error, try again.
    }
    IP_Logf_Application("New IPv6 client accepted.");
}
_Process(hSockChild);           // Process the client.
closesocket(hSockChild);        // Close connection to client from our side (too).
}
}

```

The supplied example `OS_IP_SimpleServer_IPv4_IPv6.c` is a good starting point to test the reachability of your embedded host via IPv4 and IPv6.

19.9 IPv6 resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the IPv6 modules presented in the tables below have been measured on a Cortex-M3 system. Details about the further configuration can be found in the sections of the specific example.

19.9.1 ROM usage

The resource usage of the IPv6 add-on has been measured on a Cortex-M3 system size optimization.

Addon	ROM
embOS/IP IPv6	approximately 8.00 Kbytes

Table 19.12: IPv6 ROM usage Cortex-M3

The stated ROM usage is only the additional space that is required to add IPv6 to the embOS/IP IPv4 stack. The total ROM usage for embOS/IP running IPv4 and IPv6 is approximately 28 Kbytes.

19.9.2 RAM usage

The total memory requirements of the IPv6 add-on can basically be computed as the sum of the following components:

Description	ROM
IPv6 add-on	app. 200 bytes
Unicast address	$n * \text{app. 48 bytes}$
Multicast address	$n * \text{app. 28 bytes}$
NDP entry	$n * 52 \text{ bytes}$

An IPv6 target with two unicast addresses, four Multicast address and five NDP entries needs app. 660 bytes additional RAM. For detailed information about the configuration and the memory requirements for each TCP/UDP connection, refer to *Configuring embOS/IP* on page 421.

The required memory is taken from the memory pool of the stack. For further information about how to increase the memory pool, refer to *IP_AssignMemory()* on page 59.

Chapter 20

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.

20.1 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. The standard application consists of two files. <code>OS_IP_Webserver.c</code> and <code>Webserver_DynContent.c</code> . <code>OS_IP_Webserver.c</code> fits for the most applications without modifications. <code>Webserver_DynContent.c</code> includes the dynamic parts of our sample application, like virtual files, CGI functions, etc.
.\Config\	Contains the Web server configuration file. Refer to <i>Configuration</i> on page 489 for detailed information.
.\Inc\	Contains the required include files.
.\IP\	Contains the Web server sources, <code>IP_Webserver.c</code> , <code>IP_Webserver.h</code> and <code>IP_UTIL_BASE64.c</code> , <code>IP_UTIL.h</code> .
.\IP\FS\	Contains the sources for the file system abstraction layer and the read-only file system. Refer to <i>File system abstraction layer</i> on page 818 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 <i>Using the Web server sample</i> on page 467 for detailed information.

Supplied directory structure of embOS/IP Web server package

20.2 Feature list

- Low memory footprint.
- Dynamic Web pages.
- Authentication supported.
- Forms: POST and GET support.
- Multiple connections supported.
- JavaScript supported.
- AJAX supported.
- SSE supported.
- REST 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.

20.3 Requirements

TCP/IP stack

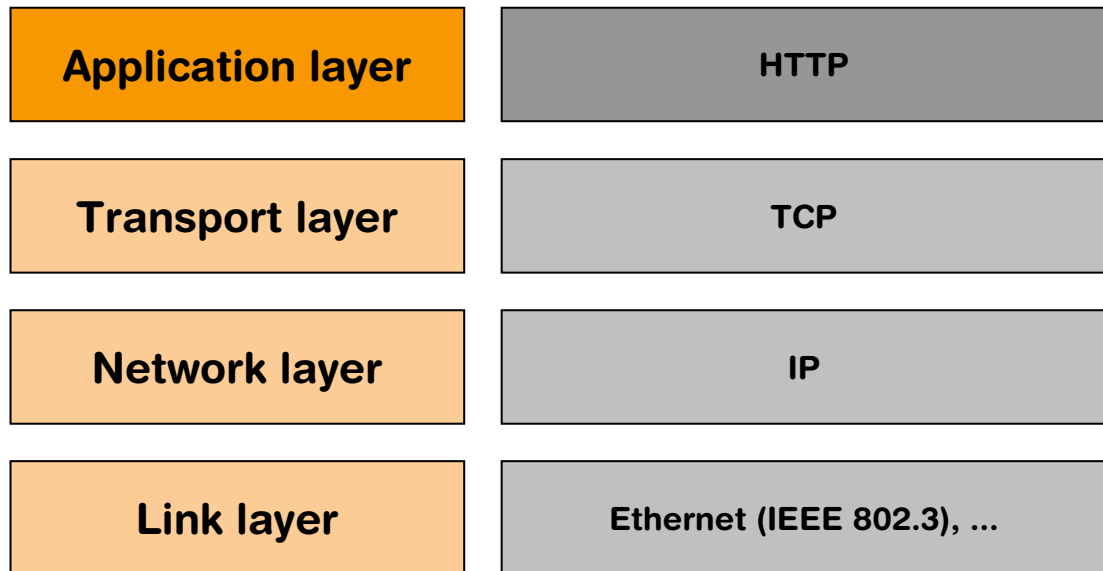
The embOS/IP Web server requires a TCP/IP stack. It is optimized for embOS/IP, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of embOS/IP.

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.

20.4 HTTP backgrounds

It is a communication protocol originally designed to transfer information via hyper-text 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.



20.4.1 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 20.1: 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 new-line, 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 signals 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
```

20.4.2 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 20.2: embOS/IP status codes

20.5 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_ProcessEx()` in a separated 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_ProcessEx()`, 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_ProcessLastEx()`, 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 818 and *File system abstraction layer* on page 818 for detailed information about the read-only file system.

20.5.1 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, a 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.

20.5.2 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 22 connections using the default configuration.

Note: 22 connections means that the target can handle up to 22 clients in parallel, if every client uses only one connection. Because a single Web browser often attempts to open more than 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 clients is less than the number of possible connections.

The 22 connections split into 20 connections that are available to be kept in the backlog of a socket (which means that up to 20 connections wait to be fetched by the application with an `accept()`) and up to 2 connections currently processed.

Every connection is handled in a separate task. Therefore, the Web server uses up to three tasks in the default configuration, one task which listens on port 80 and accepts connections and two tasks to process the accepted connections. To modify the number of connections, only the macro `MAX_CONNECTIONS` has to be modified.

The most of the supplied sample Web pages include dynamic content, refer to *Dynamic content* on page 470 for detailed information about the implementation of dynamic content.

20.5.3 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 `_WebServerParentTask()` of the example `OS_IP_Webserver.c`. Excerpt from `OS_IP_Webserver.c`:

```

/*****
*
*      _WebServerParentTask
*
*/
static void _WebServerParentTask(void) {
    struct sockaddr      Addr;
    struct sockaddr_in  InAddr;
    U32  Timeout;
    long hSockListen;
    long hSock;
    int  AddrLen;
    int  i;
    int  t;
    int  t0;
    int  r;
    WEBS_BUFFER_SIZES BufferSizes;

    Timeout = IDLE_TIMEOUT;
    IP_WEBS_SetFileInfoCallback(&_pfGetFileInfo);
    //
    // Assign file system
    //
    _pFS_API = &IP_FS_ReadOnly;    // To use a a real filesystem like emFile
                                   // replace this line.
    // _pFS_API = &IP_FS_FS;        // Use emFile

```

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;
```

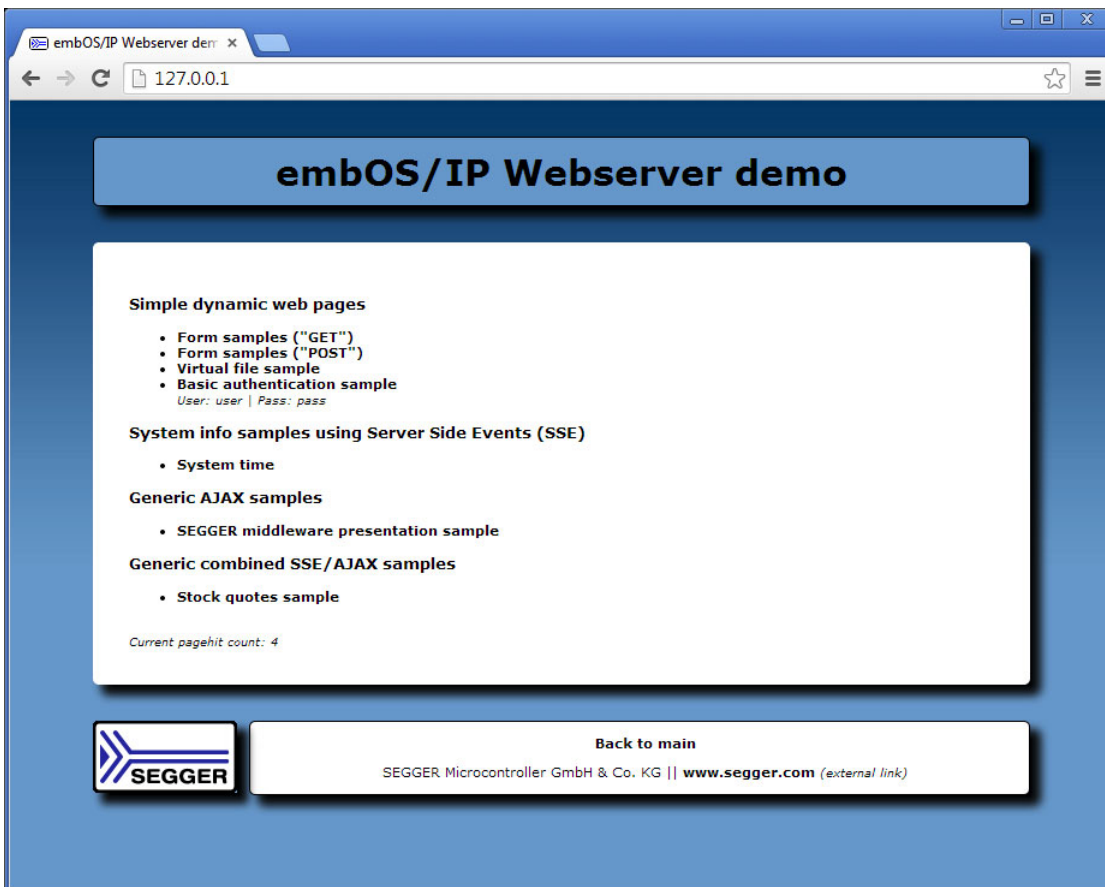
Refer to *File system abstraction layer* on page 818 and *File system abstraction layer* on page 818 for detailed information about the emFile and read-only file system abstraction layer.

20.6 Dynamic content

embOS/IP supports 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.

20.6.1 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}
};
```

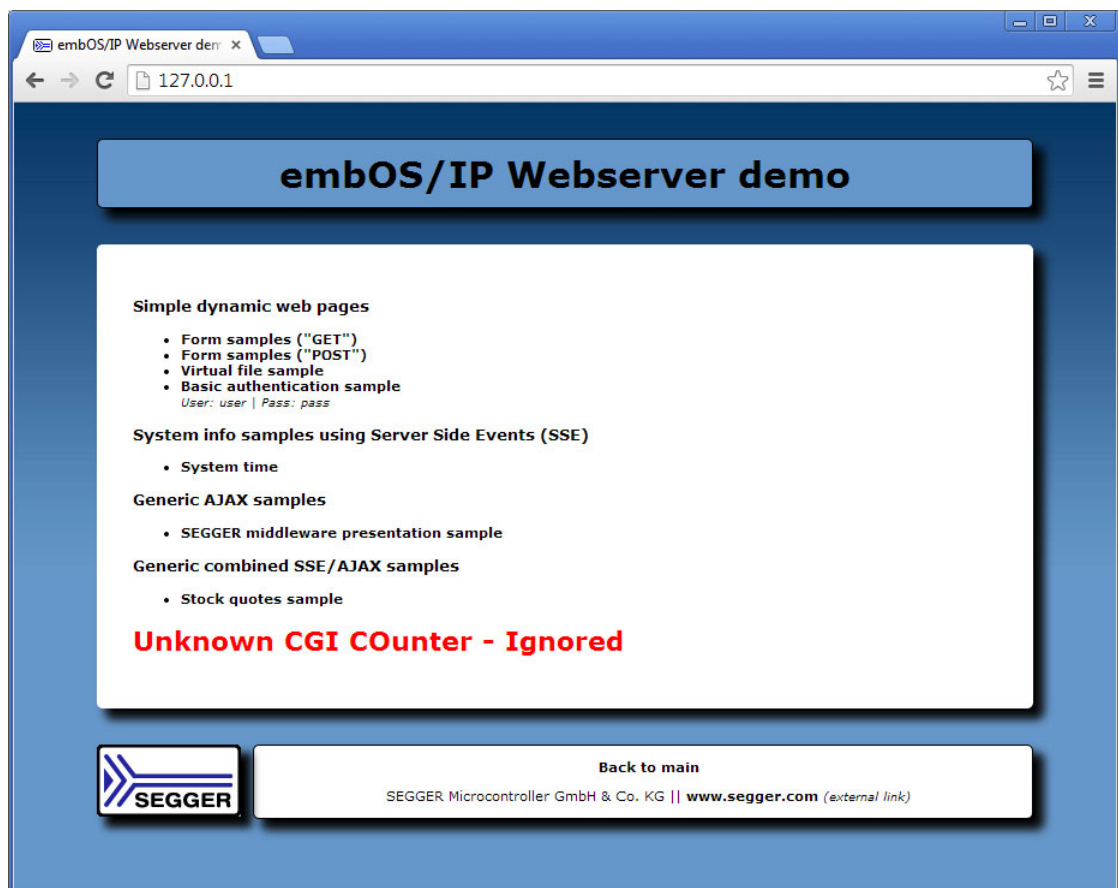
`_callback_ExecCounter()` is a simple example of how to use the CGI feature. It returns a string that includes the value of a variable which is incremented with every call to `_callback_ExecCounter()`.

```
static void _callback_ExecCounter( WEBS_OUTPUT* pOutput,
                                   const char*  sParameters,
                                   const char*  sValue ) {

    char ac[80];

    WEBS_USE_PARA(sParameters);
    WEBS_USE_PARA(sValue);
    _Cnt++;
    SEGGER_snprintf(ac, sizeof(ac), "<br><span class=\"hint\">Current page hit count:
%lu</span></ul>", _Cnt);
    IP_WEBS_SendString(pOutput, ac);
}
```

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.



20.6.1.1 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      },
    {"GetIndex"     , _callback_ExecGetIndex     },
    {"UserCGI"      , _callback_ExecUserCGI      },
    {NULL,          _callback_DefaultHandler    }
};
```

2. Implement the new function in your application source code.

```
static 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:

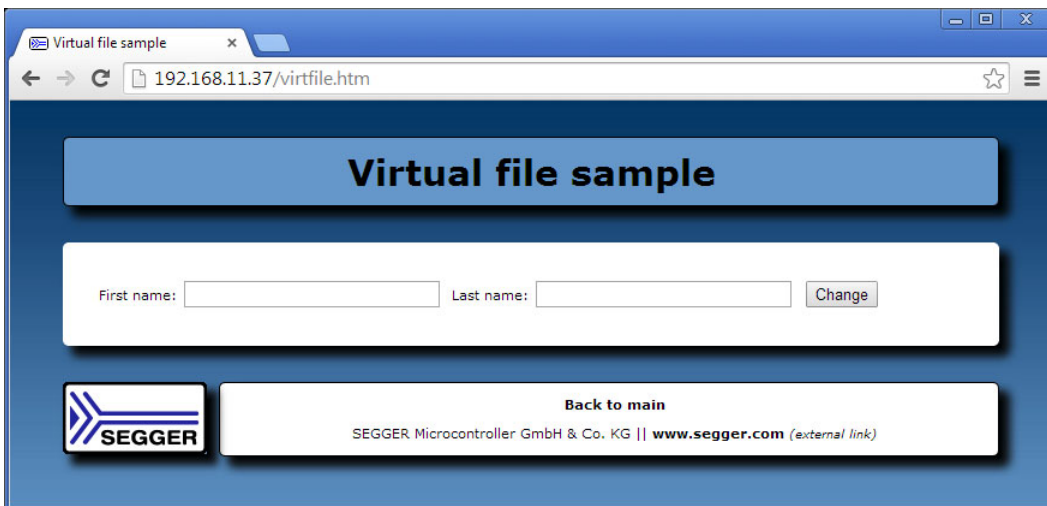
```
<!--#exec cgi="UserCGI"-->
```

20.6.2 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 521 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_CGI_Send()`.

```

/*****
 *
 *      _SendPageHeader
 *
 *  Function description:
 *      Sends the header of the virtual file.
 *      The virtual files in our sample application use the same HTML layout.
 *      The only difference between the virtual files is the content and that
 *      each of them use an own title/heading.
 */
static void _SendPageHeader(WEBS_OUTPUT * pOutput, const char * sName) {
    IP_WEBS_SendString(pOutput, "<!DOCTYPE html><html><head><title>");
    IP_WEBS_SendString(pOutput, sName);
    IP_WEBS_SendString(pOutput, "</title>");
    IP_WEBS_SendString(pOutput,
        "<link href=\"Styles.css\" rel=\"stylesheet\"></head><body><header>");
    IP_WEBS_SendString(pOutput, sName);
    IP_WEBS_SendString(pOutput, "</header>");
    IP_WEBS_SendString(pOutput, "<div class=\"content\">");
}

/*****
 *
 *      _SendPageFooter
 *
 *  Function description:
 *      Sends the footer of the virtual file.
 *      The virtual files in our sample application use the same HTML layout.
 */
static void _SendPageFooter(WEBS_OUTPUT * pOutput) {
    IP_WEBS_SendString(pOutput, "<br><br><br>");
    IP_WEBS_SendString(pOutput,
        "</div><img src=\"Logo.gif\" alt=\"Segger logo\" class=\"logo\">");
    IP_WEBS_SendString(pOutput,
        "<footer><p><a href=\"index.htm\">Back to main</a></p>");
    IP_WEBS_SendString(pOutput,
        "<p>SEGGER Microcontroller GmbH & Co. KG
        || <a href=\"http://www.segger.com\">www.segger.com</a> ");
    IP_WEBS_SendString(pOutput,
        "<span class=\"hint\">(external link)</span></p></footer></body></html>");
}

/*****
 *
 *      _callback_CGI_Send
 */
static void _callback_CGI_Send(WEBS_OUTPUT * pOutput, const char * sParameters) {
    int    r;
    const char * pFirstName;
```

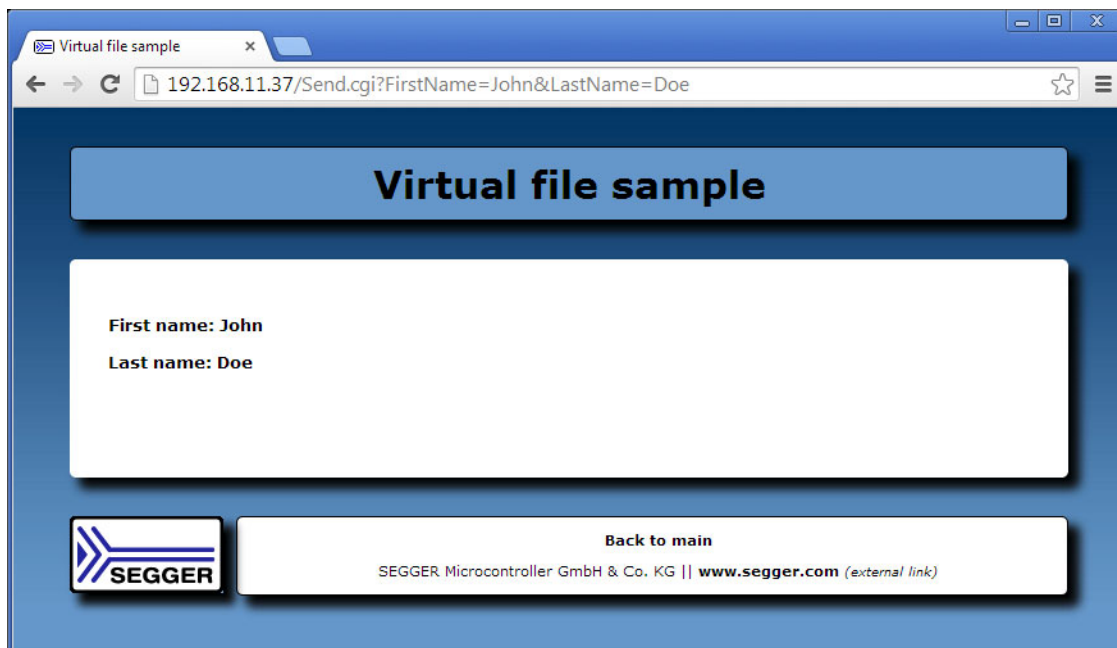
```

    int    FirstNameLen;
const char * pLastName;
    int    LastNameLen;

//
// Header of the page
//
_SendPageHeader(pOutput, "Virtual file sample");
//
// Content
//
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, "<h3>First name: ");
    IP_WEBS_SendMem(pOutput, pFirstName, FirstNameLen);
    IP_WEBS_SendString(pOutput, "</h3>");
    IP_WEBS_SendString(pOutput, "<h3>Last name: ");
    IP_WEBS_SendMem(pOutput, pLastName, LastNameLen);
    IP_WEBS_SendString(pOutput, "</h3>");
} else {
    IP_WEBS_SendString(pOutput, "<br>Error!");
}
//
// Footer of the page
//
_SendPageFooter(pOutput);
}

```

The output of `_callback_CGI_Send()` should be similar to:



20.6.3 AJAX

The embOS/IP Web server supports AJAX. AJAX is an acronym for Asynchronous JavaScript and XML. It is the foundation to build responsive and dynamic Web applications, which look and feel like desktop applications. AJAX is a special way to communicate with a Web server. In opposite to the old fashioned synchronous way where every data transmission to or from the server needs a reload of the whole Web page, AJAX works behind the scenes. With AJAX it is possible to grab the data you want and display it instantly in a Web page. No page refreshes needed, no waiting, no flickering in the browser. AJAX works on all major browsers.

AJAX combines some well-known Web techniques like HTML and CSS, JavaScript and XML. From the perspective of a developer the really interesting part AJAX are the XMLHttpRequests. An XMLHttpRequest object is an API available to Web browser scripting languages such as JavaScript and the core component for the asynchronous communication. The name XMLHttpRequest is a little bit misleading, since XMLHttpRequest can be used to send and receive any kind of data, not just XML. In most cases the exchanged data is plain text, HTML or JSON.

To exchange data without a reload of the whole page, you need create an XMLHttpRequest object. The way to create an XMLHttpRequest object is browser dependent. Therefore, it make sense to capsule the creation in a JavaScript function, which tries to handle every browser. An example is listed below:

```
//
// Create a XMLHttpRequest.
// Tries to handle the different browser implementation
//
function _CreateRequest() {
    try {
        request = new XMLHttpRequest();
    } catch (tryMS) {
        try {
            request = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (otherMS) {
            try {
                request = new ActiveXObject("Microsoft.XMLHTTP");
            } catch (failed) {
                request = null;
            }
        }
    }
    return request;
}
```

With an XMLHttpRequest object it is easy to exchange data with a Web server. The XMLHttpRequest object only needs an URI which should be requested from the server and a callback function, which will be called as soon as data will be received.

```
//
// Request the details from the server.
//
function _GetData(itemName) {
    request = _CreateRequest(); // Create an XMLHttpRequest
    if (request == null) {
        alert("Unable to create request");
        return;
    }
    var url= "../GetData.cgi";
    request.open("GET", url, true);
    request.onreadystatechange = _YourFunction;
    request.send(null);
}
```

In the example `GetData.cgi` is a virtual file, which will be requested from the embOS/IP Web server and `_YourFunction` is registered as callback function. For further information about the implementation of a virtual file, refer to *Virtual files* on page 472.

After setting the callback function, the request has to be sent. `_YourFunction` will be called each time the `readyState` of the request object changes. To guarantee that the request is complete and the transmission status is ok, the `readyState` and HTTP status should be checked in your callback function.

```
//
// Request the details from the server.
//
function _DisplayDetails() {
    if (request.readyState == 4) { // Is the request complete ?
        if (request.status == 200) { // Status OK ?
            // Do something with the received data.
        }
    }
}
```

The embOS/IP Web server comes with some sample Web pages to demonstrate how AJAX can be used to get and visualize data from your target.

File	Description
Products.htm	Simple AJAX sample. The Web page shows pictures of SEGGER middleware products. On click on one of the pictures an XMLHttpRequest is created and additional product related information is requested from the embOS/IP Web server.
Shares.htm	The sample demonstrates the usage of Server-Sent Events (SSE) and AJAX with the embOS/IP Web server. The displayed stock quotes table is random data generated by the server and updated every second via SSE. The graph is updated via AJAX. Every update of the stock prices table triggers the graph library to request the last 30 stock prices of the selected company to redraw the graph. All stock quotes are fictional. The goal of this example is to demonstrate how simple it is to visualize any kind of data with the embOS/IP Web server. The sample uses the open source JavaScript library RGraph for the visualization of the stock quotes. RGraph is an HTML5 charts library, which uses the MIT license. The latest version of the library can be downloaded from http://www.rgraph.net/ .

Table 20.3: embOS/IP Web server sample application using AJAX

All samples are hardware independent and tested with popular Web browsers like Internet Explorer, Firefox and Chrome.

For further information about AJAX, XMLHttpRequest handling, and data visualization we recommend one of the many available reference books.

20.6.4 Server-Sent Events (SSE)

Server-Sent Events (SSE) are an HTML5 technology which enables a Web server to push data to Web pages over HTTP. The Web browser establishes an initial connection, which is used for the Web server updates.

The embOS/IP Web server supports SSE to supply the Web browser with dynamic content. A Web browser can request information via EventSource objects. The idea behind SSE is that the Web server keeps an connection to Web browser open and sends data via this connection whenever it is necessary. This means that the Web browser receives data as a stream without polling. This reduces the HTTP protocol overhead.

To subscribe to an event stream, you have create an EventSource object and pass it the URL of your stream.

```
<script>
  if(typeof(EventSource) !== "undefined") {
    var source = new EventSource("SSETime.cgi");
    source.onmessage = function(event) {
      document.getElementById("Time").innerHTML = "<h2>" + event.data + "</h2>";
      document.getElementById("Time").innerHTML +=
        "The browser gets the system time via a Server-Sent Event (SSE).
        <br>No meta refresh (reload) required!";
    };
  } else {
    document.getElementById("Time").innerHTML =
      "Sorry, your browser does not support Server-Sent Events (SSE)...";
  }
</script>
```

The source code excerpt above creates a new EventSource object. The EventSource objects starts immediately listening for events on the given URL `SSETime.cgi`. Every-time when the Web browser will receive data, the data will be displayed on the Web page.

`SSETime.cgi` is implemented in the supplied embOS/IP Web Server sample application (`Webserver_DynContent.c`). From the perspective of the Web server it is a virtual file. Please refer to *Virtual files* on page 472 for further information about virtual files. The following excerpt of `Webserver_DynContent.c` shows the implementation of the virtual file `SSETime.cgi`.

```
/* *****
 *
 *      _callback_CGI_SSETime
 *
 *  Function description:
 *      Sends the system time to the Web browser every 500 ms.
 */
static void _callback_CGI_SSETime(WEB_OUTPUT * pOutput, const char * sParameters) {
    int r;

    IP_USE_PARA(sParameters);
    //
    // Construct the SSE Header
    //
    IP_WEBS_SendHeaderEx(pOutput, NULL, "text/event-stream", 1);
    IP_WEBS_SendString(pOutput, "retry: 1000\n");
    while(1) {
        r = _SendTime((void*)pOutput);
        if (r == 0) { // Data transmitted
            OS_Delay(500);
        } else {
            break;
        }
    }
}
```

First step to send data as an event stream is to send the MIME type `text/event-stream` to the Web browser. The Web browser attempts to reconnect to the Web server ~3 seconds after a connection is closed. You can change that timeout by sending a line beginning with `retry:`, followed by the number of milliseconds to wait

before trying to reconnect. The event stream message is build in `_SendTime()`. After sending the data `OS_Delay()` suspends the task for 500ms. `_SendTime()` will be called again after the delay as long as the connection is open.

```

/*****
 *
 *      _SendTime
 *
 *
 *  Function description:
 *      Sends the system time to the Web browser.
 *
 *  Return value:
 *      0: Data successfully sent.
 *      -1: Data not sent.
 *      1: Data successfully sent. Connection should be closed.
 */
static int _SendTime(WEBS_OUTPUT * pOutput) {
    int r;

    //
    // Send implementation specific header to client
    //
    IP_WEBS_SendString(pOutput, "data: ");
    IP_WEBS_SendString(pOutput, "System time: ");
    IP_WEBS_SendUnsigned(pOutput, OS_GetTime32(), 10, 0);
    IP_WEBS_SendString(pOutput, "<br>");
    IP_WEBS_SendString(pOutput, "\n\n");          // End of the SSE data
    r = IP_WEBS_Flush(pOutput);
    return r;
}

```

The return value of `_SendTime()` is checked in `_callback_CGI_SSETime()`. This is necessary to prove if the data connection is still open. If the connection has been closed by the client, the endless loop will be left and the Web server will end the Web server child task.

SSE is currently not support by all popular browsers. The following Web browsers support Server-Sent Events natively.

Browser	Supported	Notes
Google Chrome	Yes	Starting with Chrome Ver. 27
MS Internet Explorer	No	--
Mozilla Firefox	Yes	Starting with Firefox Ver. 30
Opera	Yes	Starting with Ver. 23
Safari	Yes	Starting with Ver. 5.1

Table 20.4: Web browser support for Server-Sent Events

Since the Microsoft Internet Explorer does not support Server-Sent Events, we use the JavaScript library `eventsources.js` in our samples to make them also usable with Microsoft Internet Explorer. `eventsources.js` can be downloaded with the following link: <https://github.com/Yaffle/EventSource/>

`eventsources.js` uses the MIT license. It can be used and modified according to your needs.

The embOS/IP Web server comes with some sample Web pages to demonstrate how SSE can be used to get and visualize data from your target.

File	Description
Shares.htm	The sample demonstrates the usage of Server-Sent Events (SSE) and AJAX with the embOS/IP Web server. For detailed information about the sample refer to <i>AJAX</i> on page 475.
SSE_IP.htm	Shows some embOS/IP status information.
SSE_OS.htm	Shows some embOS status information.
SSE_Time.htm	Shows the system time.

Table 20.5: embOS/IP Web server sample application using SSE

20.7 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
```


20.7.1 Authentication example

The client requests a resource for which authentication is required:

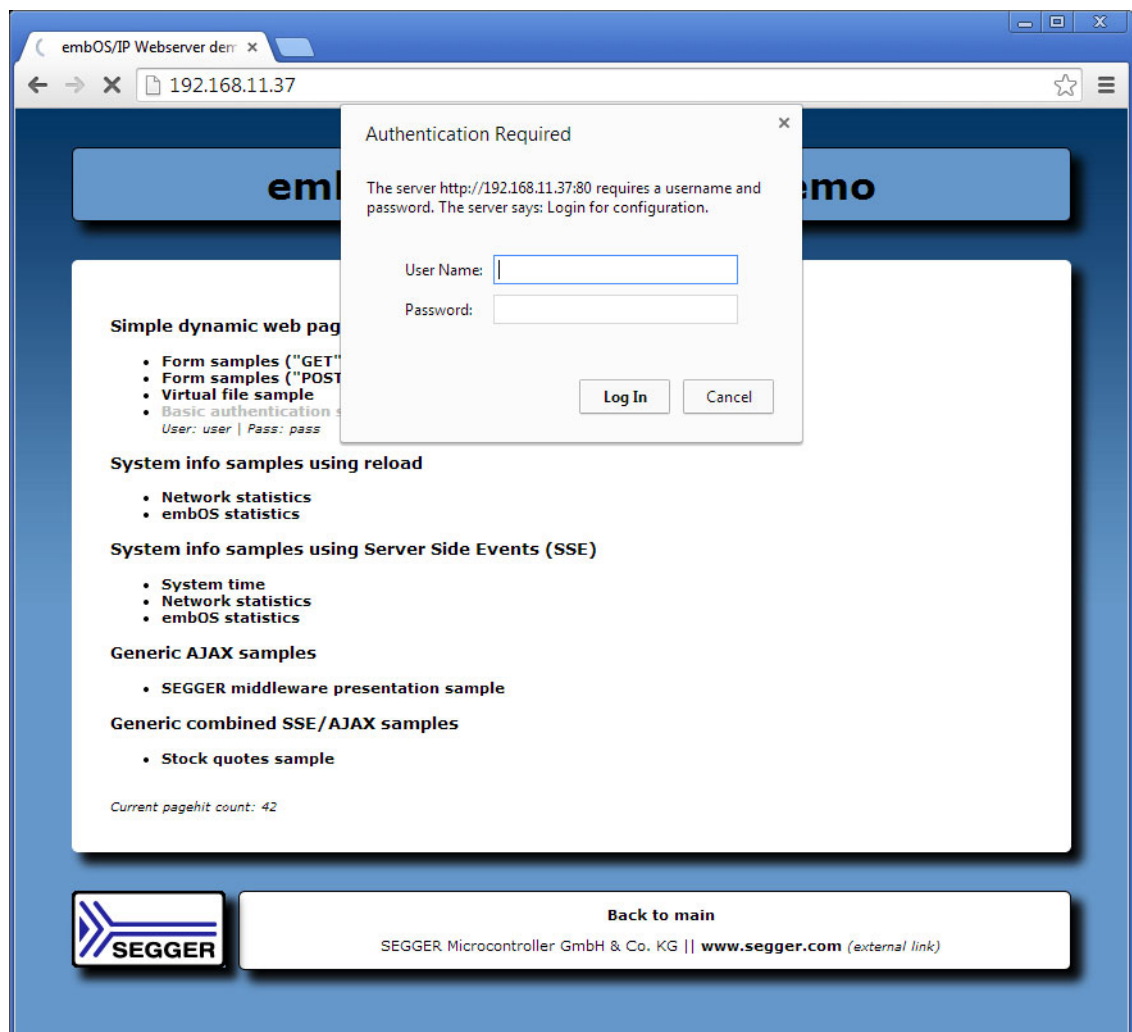
```
GET /conf/Authen.htm HTTP/1.1
Host: 192.168.11.37
```

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: 192.168.11.37
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
```

20.7.2 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 548 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.

20.8 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
- . _ - *()!\$\\

20.8.1 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 `FormGET.htm` implements a form with three inputs, two text fields and one button.

An excerpt of the HTML code of the Web page as it is added to the server is listed below:

```
<hr>
<h2>Please enter your name...</h2>
<h3>Hello <!--#exec cgi="FirstName"--> <!--#exec cgi="LastName"-->!</h3>
<form action="" method="GET">
  <label for="FirstName">First name: </label>
  <input name="FirstName" type="text" size="30" maxlength="30"
    value="<!--#exec cgi="FirstName"-->">&nbsp;
  <label for="LastName">Last name: </label>
  <input name="LastName" type="text" size="30" maxlength="30"
    value="<!--#exec cgi="LastName"-->">&nbsp;
  <input type="submit" value="Change">
</form>
<hr>
```

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:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Virtual file sample</title>
    <link href="Styles.css" rel="stylesheet">
  </head>
  <body>
    <header>Virtual file sample</header>
    <div class="content">
      <form action="Send.cgi" method="GET">
        <label for="FirstName">First name: </label>
        <input name="FirstName" type="text" size="30" maxlength="30" value="">&nbsp;
        <label for="LastName">Last name: </label>
        <input name="LastName" type="text" size="30" maxlength="30" value="">&nbsp;
        <input type="submit" value="Change">
      </form>
    </div>
    
    <footer>
      <p>
        <a href="index.htm">
          Back to main</a></p>
        <p>SEGGER Microcontroller GmbH & Co. KG ||
        <a href="http://www.segger.com">www.segger.com</a>
        <span class="hint">(external link)</span>
      </p>
    </footer>
  </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://192.168.11.37/FormGET.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 string Hello John Doe above the input fields until you enter and transmit another name to the server.

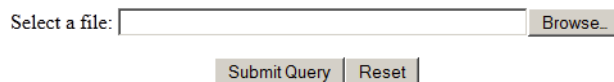
20.9 File upload

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 483. 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 535 or by using `IP_WEBS_GetParaValuePtr()` on page 536.

20.9.1 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.

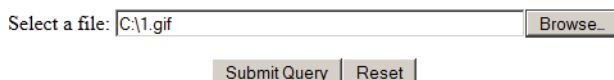


The HTML code of the Web page as it is added to the server is listed below:

```
<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 483. 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 provided 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>

```


20.10 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.

20.10.1 Compile time 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).

20.10.2 Compile time configuration switches

Type	Symbolic name	Default	Description
F	<code>WEBS_WARN</code>	--	Defines a function to output warnings. In debug configurations (<code>DEBUG == 1</code>) <code>WEBS_WARN</code> maps to <code>IP_Warnf_Application()</code> .
F	<code>WEBS_LOG</code>	--	Defines a function to output logging messages. In debug configurations (<code>DEBUG == 1</code>) <code>WEBS_LOG</code> maps to <code>IP_Logf_Application()</code> .
N	<code>WEBS_IN_BUFFER_SIZE</code>	512	Defines the size of the input buffer. The input buffer is used to store the HTTP client requests. Please refer to <i>Runtime configuration</i> on page 491 for further information about the usage.
N	<code>WEBS_OUT_BUFFER_SIZE</code>	512	Defines the size of the output buffer. The output buffer is used to store the HTTP response. Please refer to <i>Runtime configuration</i> on page 491 for further information about the usage.

Type	Symbolic name	Default	Description
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. Please refer to <i>Runtime configuration</i> on page 491 for further information about the usage.
N	WEBS_FILENAME_BUFFER_SIZE	64	Defines the size of the buffer used to store the requested URI/filename to access on the filesystem. Please refer to <i>Runtime configuration</i> on page 491 for further information about the usage.
N	WEBS_TEMP_BUFFER_SIZE	256	Defines the size of the TEMP buffer used internally by the Web server.
N	WEBS_AUTH_BUFFER_SIZE	32	Defines the size of the buffer used to store the authentication string. Refer to <i>Authentication</i> on page 480 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. If you do not use the upload feature, define WEBS_SUPPORT_UPLOAD to 0.
N	WEBS_URI_BUFFER_SIZE	0	Defines the size of the buffer used to store the "full URI" of the accessed resource. By default this feature is disabled.
N	WEBS_MAX_ROOT_PATH_LEN	12	Maximum allowed root path length of the Web server in multiples of a CPU native unit (typically int). If the root path of the Web server is the root of your media you can comment out this define or set it to zero. Example: For the root path "/httpdocs" the define needs to be set to at least 9 . As this is not a multiple of an int, set it to 12.

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	<pre>"<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"</pre>
A	WEBS_404_PAGE	<pre>"<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"</pre>
A	WEBS_501_PAGE	<pre>"<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"</pre>
A	WEBS_503_PAGE	<pre>"<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"</pre>

20.10.3 Runtime configuration

The input buffer, the output buffer, the parameter buffer, the filename buffer and the maximum root path length are runtime configurable. Up to version 2.20h compile time switches WEBS_IN_BUFFER_SIZE, WEBS_OUT_BUFFER_SIZE and WEBS_PARA_BUFFER_SIZE were used to configure the sizes of the buffers. These compile time switches along with new switches like WEBS_MAX_ROOT_PATH_LEN are still available to guarantee compatibility to previous versions and are used as default values for the buffer sizes in applications where the runtime configuration function IP_WEBS_ConfigBufSizes() is not called. For further information, please refer to IP_WEBS_ConfigBufSizes() on page 498.

20.11 API functions

Function	Description
<code>IP_WEBS_AddUpload()</code>	Adds the upload functionality to the Web server.
<code>IP_WEBS_AddFileTypeHook()</code>	Adds a new file name extension to MIME type correlation.
<code>IP_WEBS_AddPreContentOutputHook()</code>	This function registers a hook with a callback that is called before sending content.
<code>IP_WEBS_AddRequestNotifyHook()</code>	Registers a hook to be notified upon an incoming request.
<code>IP_WEBS_ConfigBufSizes()</code>	Sets the sizes of the required buffers.
<code>IP_WEBS_ConfigRootPath()</code>	Sets the root path in filesystem to use.
<code>IP_WEBS_ConfigUploadRootPath()</code>	Sets the upload root path in filesystem to use.
<code>IP_WEBS_Flush()</code>	Flushes the output buffer.
<code>IP_WEBS_Init()</code>	Initializes a child context.
<code>IP_WEBS_OnConnectionLimit()</code>	Outputs an error message to the connected client.
<code>IP_WEBS_Process()</code>	Processes a HTTP request of a client.
<code>IP_WEBS_ProcessEx()</code>	Processes a HTTP request of a client.
<code>IP_WEBS_ProcessLast()</code>	Processes a HTTP request of a client and closes the connection thereafter.
<code>IP_WEBS_ProcessLastEx()</code>	Processes a HTTP request of a client and closes the connection thereafter.
<code>IP_WEBS_Redirect()</code>	Redirect to a file on file system by sending its content.
<code>IP_WEBS_Reset()</code>	Resets internal structures.
<code>IP_WEBS_RetrieveUserContext()</code>	Retrieves a previously stored user context from the current connection context.
<code>IP_WEBS_SendHeader()</code>	Sends a header with data provided.
<code>IP_WEBS_SendHeaderEx()</code>	Sends a header with data provided.
<code>IP_WEBS_SendLocationHeader()</code>	Sends a header with a redirection code for the browser.
<code>IP_WEBS_SendMem()</code>	Sends data to a connected target.
<code>IP_WEBS_SendString()</code>	Sends a string to a connected target.
<code>IP_WEBS_SendStringEnc()</code>	Encodes and sends a string to a connected target.
<code>IP_WEBS_SendUnsigned()</code>	Sends an unsigned value to a connected target.
<code>IP_WEBS_SetFileInfoCallback()</code>	Sets a callback function to handle file information used by the Web server.
<code>IP_WEBS_SetHeaderCacheControl()</code>	Sets the string to be sent for the cache-control field in the header.
<code>IP_WEBS_StoreUserContext()</code>	Saves an user context into the current connection context.
CGI/virtual file related functions	
<code>IP_WEBS_AddVFileHook()</code>	Adds a hook to serve a simple virtual file.
<code>IP_WEBS_CompareFileNameExt()</code>	Checks the file name extension.
<code>IP_WEBS_ConfigSendVFileHeader()</code>	Configures automatic sending of a header based on the file name for virtual files.
<code>IP_WEBS_ConfigSendVFileHookHeader()</code>	Configures automatic sending of a header based on the file name for VFile hooks.

Table 20.6: embOS/IP Web server interface function overview

Function	Description
<code>IP_WEBS_DecodeAndCopyStr()</code>	Decodes an HTML encoded string and copy it into a buffer.
<code>IP_WEBS_DecodeString()</code>	Decodes an HTML encoded string.
<code>IP_WEBS_GetDecodedStrLen()</code>	Returns the length of a HTML encoded string after decoding.
<code>IP_WEBS_GetNumParas()</code>	Returns the number of parameter/value pairs.
<code>IP_WEBS_GetParaValue()</code>	Gets a parameter value pair.
<code>IP_WEBS_GetParaValuePtr()</code>	Gets a parameter value pairs pointers for further processing.
<code>IP_WEBS_GetConnectInfo()</code>	Returns the connect info passed to the Web Server upon start.
<code>IP_WEBS_GetURI()</code>	Returns the URI of the accessed resource.
<code>IP_WEBS_UseRawEncoding()</code>	Use RAW encoding instead of auto detection when required.
METHOD extension related functions	
<code>IP_WEBS_METHOD_AddHook()</code>	Adds a new METHOD hook.
<code>IP_WEBS_METHOD_CopyData()</code>	Retrieves data sent from within a METHOD hook callback.
Utility functions	
<code>IP_UTIL_BASE64_Decode()</code>	Decodes a Base64 encoded string.
<code>IP_UTIL_BASE64_Encode()</code>	Encodes a string as a Base64 string.

Table 20.6: embOS/IP Web server interface function overview (Continued)

20.11.1 IP_WEBS_AddUpload()

Description

Adds the upload functionality to the Web server.

Prototype

```
int    IP_WEBS_AddUpload ( void );
```

Return value

1: OK. Upload enabled

0: Upload could not be enabled. (WEBS_SUPPORT_UPLOAD == 0)

Additional information

A real file system like emFile is required to upload files. Calling this function has no effect if the compile time switch WEBS_SUPPORT_UPLOAD is not defined to 1. In library versions of the Web server WEBS_SUPPORT_UPLOAD is always defined to 1.

20.11.2 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
<code>pHook</code>	[IN] Pointer to an element of type <code>WEBS_FILE_TYPE_HOOK</code> .
<code>sExt</code>	[IN] String containing the extension without leading dot.
<code>sContent</code>	[IN] String containing the MIME type associated to the extension.

Table 20.7: 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 unknown file format. The header sent for this file in case a client requests it will be generated based on this information. Refer to *Structure WEBS_FILE_TYPE_HOOK* on page 554 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");
}
```

20.11.3 IP_WEBS_AddPreContentOutputHook()

Description

This function registers a hook with a callback that is called before sending content.

Prototype

```
void IP_WEBS_AddPreContentOutputHook( WEBS_PRE_CONTENT_OUTPUT_HOOK* pHook,
                                      IP_WEBS_pfPreContentOutput    pf,
                                      U32                             Mask );
```

Parameter

Parameter	Description
<code>pHook</code>	Pointer to an element of type <code>WEBS_PRE_CONTENT_OUTPUT_HOOK</code> .
<code>pf</code>	Callback to execute before the Web server sends content to its best knowledge.

Table 20.8: IP_WEBS_AddPreContentOutputHook() parameter list

Additional information

Refer to *Structure WEBS_PRE_CONTENT_OUTPUT_HOOK* on page 556 for detailed information about the structure `WEBS_PRE_CONTENT_OUTPUT_HOOK`.

This hook can be used to intercept the web server right before sending content to its best knowledge. As all form data has been processed at this time this can be used to decide if all form data is valid like for a user:pass combination processed via a form. For example upon invalid login data it is possible to redirect from within the callback back to the login page using *IP_WEBS_Redirect()* on page 511.

20.11.4 IP_WEBS_AddRequestNotifyHook()

Description

This function registers a hook with a callback that is called on each request to pass some information like URI and METHOD used to the application.

Prototype

```
void IP_WEBS_AddRequestNotifyHook ( WEBS_REQUEST_NOTIFY_HOOK* pHook,
                                     IP_WEBS_pfRequestNotify   pf );
```

Parameter

Parameter	Description
pHook	Pointer to an element of type WEBS_REQUEST_NOTIFY_HOOK.
pf	Callback to execute upon a request to the Web server.

Table 20.9: IP_WEBS_AddRequestNotifyHook() parameter list

Additional information

Refer to *Structure WEBS_REQUEST_NOTIFY_HOOK* on page 557 for detailed information about the structure WEBS_REQUEST_NOTIFY_HOOK.

20.11.5 IP_WEBS_ConfigBufSizes()

Description

Configures the sizes of the buffers used by the Web server.

Prototype

```
void IP_WEBS_ConfigBufSizes ( WEBS_BUFFER_SIZES *pBufSizes );
```

Parameter

Parameter	Description
pBufSizes	[IN] Structure holding the sizes of the required buffers.

Table 20.10: IP_WEBS_ConfigBufSizes() parameter list

Additional information

The structure WEBS_BUFFER_SIZES is defined as follow:

```
typedef struct WEBS_BUFFER_SIZES {  
    U32 NumBytesInBuf;           // Size of the input buffer. By default: 256 bytes  
    U32 NumBytesOutBuf;          // Size of the output buffer. By default: 512 bytes  
    U32 NumBytesFilenameBuf;     // Size of the output buffer. By default: 512 bytes  
    U32 MaxRootPathLen;          // Size of the output buffer. By default: 512 bytes  
    U32 NumBytesParaBuf;         // Size of the parameter buffer. By default: 64 bytes  
} WEBS_BUFFER_SIZES;
```

Since version 3.00, the buffers used by the Web server are runtime configurable. Earlier versions of the Web server used compile time switches to define the buffer sizes. If IP_WEBS_ConfigBufSizes() will not be called, the values of the compile time switches will be used to configure the buffer sizes.

We recommend at least 256 bytes for the input buffer and 512 bytes for the output buffer. If virtual files are not used in your application, the parameter buffer and others can be set to 0 to save RAM.

20.11.6 IP_WEBS_ConfigRootPath()

Description

Configures the internal root path to prepend to the requested URI.

Prototype

```
int IP_WEBS_ConfigRootPath( const char* sRootPath );
```

Parameter

Parameter	Description
sRootPath	[IN] String to root path to prepend such as "/httpdocs".

Table 20.11: IP_WEBS_ConfigRootPath() parameter list

Return value

O.K.: 0

Error: 1

Additional information

By default the root path used is the root path of your filesystem. Configuring a root path can be used to separate the Web server from other services like an FTP server root folder. A classic use sample would be that all Web pages are stored in the subfolder "/httpdocs". In this case you can call `IP_WEBS_ConfigRootPath(/httpdocs)` to load all Web pages relative from this folder in your filesystem instead from the root folder of your filesystem.

Other services like an FTP server can be used to grant access to the root folder of your filesystem to allow access to the `/httpdocs` subfolder and other files in your filesystem as well.

The root path can be as long as the maximum root path length configured. If `IP_WEBS_ConfigBufSizes()` will not be called to set the max. root path length, the default `WEBS_MAX_ROOT_PATH_LEN` will be used.

20.11.7 IP_WEBS_ConfigUploadRootPath()

Description

Configures the internal root path to prepend to the requested URI.

Prototype

```
int IP_WEBS_ConfigUploadRootPath( const char* sUploadRootPath );
```

Parameter

Parameter	Description
<code>sRootPath</code>	[IN] String to root path to prepend to the upload location in the local filesystem like "/upload".

Table 20.12: IP_WEBS_ConfigUploadRootPath() parameter list

Return value

O.K.: 0

Error: 1

Additional information

By default uploads are placed in the root folder of your filesystem. Configuring an upload root path can be used to redirect uploads to another path like an "/upload" folder.

The upload root path can be as long as the maximum path that can be used with the upload filename defined by `WEBS_UPLOAD_FILENAME_BUFFER_SIZE` (minus 1 byte for string termination).

20.11.8 IP_WEBS_Flush()

Description

Sends the output buffer.

Prototype

```
int IP_WEBS_Flush ( WEBS_OUTPUT *pOutput );
```

Additional information

Normally, the stack handles all the data transmission. `IP_WEBS_Flush()` should only be used in special use cases like implementing Server-Sent Events, where data transmission should be done immediately.

Return value

- 1: Data sent. Connection will be closed after transmission.
- 0: Data sent. Connection will be kept open after transmission.
- 1: Error. Data could not be sent. Connection closed or will be closed from the stack.

20.11.9 IP_WEBS_Init()

Description

Initializes the Web server application context.

Prototype

```
void IP_WEBS_Init (      WEBS_CONTEXT      *pContext,
                        const WEBS_IP_API    *pIP_API,
                        const WEBS_SYS_API    *pSYS_API,
                        const IP_FS_API       *pFS_API,
                        const WEBS_APPLICATION *pApplication );
```

Parameter

Parameter	Description
<code>pContext</code>	[OUT] Application specific Web server context.
<code>pIP_API</code>	[IN] Pointer to a structure holding the IP related API functions.
<code>pSYS_API</code>	[IN] Pointer to a structure holding the system related API functions.
<code>pFS_API</code>	[IN] Pointer to a structure holding the file system related API functions.
<code>pApplication</code>	[IN] Pointer to a structure holding the application related structures to handle CGIs, virtual files, access control, etc.

Table 20.13: IP_WEBS_Init() parameter list

Additional information

The parameter `pContext` is a structure holding all the required function pointers to the routines used to send and receive bytes from/to the client, access the file system, allocate and free memory, etc.

```
typedef struct WEBS_CONTEXT {
    const WEBS_IP_API      *pIP_API;
    const WEBS_SYS_API      *pSYS_API;
    const IP_FS_API         *pFS_API;
    const WEBS_APPLICATION *pApplication;
    void                    *pWebsPara;
    void                    *pUpload;
} WEBS_CONTEXT;
```

`WEBS_IP_API` includes all functions, which are required for the used IP stack and is defined as follow:

```
typedef struct WEBS_IP_API {
    IP_WEBS_tSend    pfSend;
    IP_WEBS_tReceive pfReceive;
} WEBS_IP_API;
```

```
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.

WEBS_SYS_API includes all functions, which are required to allocate and free memory and is defined as follow:

```
typedef struct WEBS_SYS_API {
    IP_WEBS_tAlloc    pfAlloc;
    IP_WEBS_tFree     pfFree;
} WEBS_SYS_API;

typedef void *(*IP_WEBS_tAlloc) (          U32    NumBytesReq );

typedef void  (*IP_WEBS_tFree)  (          void   *p);
```

The alloc function returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

For details about the parameter [pFS_API](#) and the IP_FS_API structure, refer to *File system abstraction layer* on page 818. For details about the parameter [pApplication](#) and the WEBS_APPLICATION structure, refer to *Structure WEBS_APPLICATION* on page 549.

The [pWebsPara](#) and [pUpload](#) should not be changed. The stack fills the structures, if necessary.

IP_WEBS_Init() has to be called if IP_WEBS_ProccessEx() and IP_WEBS_ProcessLastEx() are used for the connection handling. Refer to *IP_WEBS_ProcessEx()* on page 508 and *IP_WEBS_ProcessLastEx()* on page 510 for detailed information.

Example

```
/* Excerpt from OS_IP_Webserver.c*/

/*****
 *
 *      _WebServerChildTask
 *
 */
static void _WebServerChildTask(void *pContext) {
    WEBS_CONTEXT ChildContext;
    long hSock;
    int Opt;

    hSock    = (long)pContext;
    Opt      = 1;
    setsockopt(hSock, SOL_SOCKET, SO_KEEPAIVE, &Opt, sizeof(Opt));
    //
    // Initialize the context of the child task.
    //
    IP_WEBS_Init(&ChildContext, &Webs_IP_API, &FS_API, &Application);
    if (_ConnectCnt < MAX_CONNECTIONS) {
        IP_WEBS_ProcessEx(&ChildContext, pContext, NULL);
    } else {
        IP_WEBS_ProcessLastEx(&ChildContext, pContext, NULL);
    }
    _closesocket(hSock);
    OS_EnterRegion();
    _AddToConnectCnt(-1);
    OS_Terminate(0);
    OS_LeaveRegion();
}
```

```

/*****
*
*      _WebServerParentTask
*
*/
static void _WebServerParentTask(void) {
    struct sockaddr    Addr;
    struct sockaddr_in InAddr;
    U32    Timeout;
    long   hSockListen;
    long   hSock;
    int     AddrLen;
    int     i;
    int     t;
    int     t0;
    WEBS_IP_API   Webs_IP_API;
    WEBS_SYS_API  Webs_SYS_API;
    WEBS_BUFFER_SIZES BufferSizes;

    Timeout = IDLE_TIMEOUT;
    IP_WEBS_SetFileInfoCallback(&_pfGetFileInfo);
    //
    // Assign file system
    //
    _pFS_API = &IP_FS_ReadOnly; // To use a a real filesystem like emFile
                                // replace this line.
    // _pFS_API = &IP_FS_FS;      // Use emFile
    // IP_WEBS_AddUpload();       // Enable upload
    //
    // Configure buffer size.
    //
    IP_MEMSET(&BufferSizes, 0, sizeof(BufferSizes));
    BufferSizes.NumBytesInBuf      = WEBS_IN_BUFFER_SIZE;
    //
    // Use max. MTU configured for the last interface added minus worst
    // case IPv4/TCP/VLAN headers. Calculation for the memory pool
    // is done under assumption of the best case headers with - 40 bytes.
    //
    BufferSizes.NumBytesOutBuf      = IP_TCP_GetMTU(_IFaceId) - 72;
    BufferSizes.NumBytesParaBuf     = WEBS_PARA_BUFFER_SIZE;
    BufferSizes.NumBytesFilenameBuf = WEBS_FILENAME_BUFFER_SIZE;
    BufferSizes.MaxRootPathLen     = WEBS_MAX_ROOT_PATH_LEN;
    //
    // Configure the size of the buffers used by the Webserver child tasks.
    //
    IP_WEBS_ConfigBufSizes(&BufferSizes);
    //
    // Give the stack some more memory to enable the dynamical memory
    // allocation for the Web server child tasks
    //
    IP_AddMemory(_aPool, sizeof(_aPool));
    //
    // Get a socket into listening state
    //
    hSockListen = socket(AF_INET, SOCK_STREAM, 0);
    if (hSockListen == SOCKET_ERROR) {
        while(1); // This should never happen!
    }
    memset(&InAddr, 0, sizeof(InAddr));
    InAddr.sin_family      = AF_INET;
    InAddr.sin_port        = htons(80);
    InAddr.sin_addr.s_addr = INADDR_ANY;
    bind(hSockListen, (struct sockaddr *)&InAddr, sizeof(InAddr));
    listen(hSockListen, BACK_LOG);
    //
    // Loop once per client and create a thread for the actual server
    //

```



```

do {
Next:
    //
    // Wait for an incoming connection
    //
    hSock = 0;
    AddrLen = sizeof(Addr);
    if ((hSock = accept(hSockListen, &Addr, &AddrLen)) == SOCKET_ERROR) {
        continue;    // Error
    }
    //
    // Create server thread to handle connection.
    // If connection limit is reached, keep trying for some time before giving up
    // and outputting an error message
    //
    t0 = OS_GetTime32() + 1000;
    do {
        if (_ConnectCnt < MAX_CONNECTIONS) {
            for (i = 0; i < MAX_CONNECTIONS; i++) {
                U8 r;
                r = OS_IsTask(&_aWebTasks[i]);
                if (r == 0) {
                    setsockopt(hSock, SOL_SOCKET, SO_RCVTIMEO, &Timeout, sizeof(Timeout));
                    OS_CREATETASK_EX(&_aWebTasks[i], "Webserver Child", _WebServerChildTask,
                                    TASK_PRIO_WEBS_CHILD, _aWebStacks[i], (void *)hSock);
                    _AddToConnectCnt(1);
                    goto Next;
                }
            }
        }
        //
        // Check time out
        //
        t = OS_GetTime32();
        if ((t - t0) > 0) {
            IP_WEBS_OnConnectionLimit(_Send, _Recv, (void*)hSock);
            _closesocket(hSock);
            break;
        }
        OS_Delay(10);
    } while(1);
} while(1);
}

```

20.11.10 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
<code>pIP_API</code>	[IN] Pointer to a structure of type <code>IP_FTPS_API</code> .
<code>CtrlSock</code>	[IN] Pointer to the socket which is related to the command connection.

Table 20.14: `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 507 and `IP_WEBS_ProcessLast()` on page 509 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();
```

20.11.11 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 20.15: IP_WEBS_Process() parameter list

Return value

==0: O.K.

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 818. For details about the parameter [pApplication](#) and the `WEBS_APPLICATION` structure, refer to *Structure WEBS_APPLICATION* on page 549.

Refer to *IP_WEBS_ProcessLast()* on page 509 and *IP_WEBS_OnConnectionLimit()* on page 506 for further information.

20.11.12 IP_WEBS_ProcessEx()

Description

Processes a HTTP request of a client.

Prototype

```
int IP_WEBS_ProcessEx ( WEBS_CONTEXT* pContext,
                        void*          pConnectInfo
                        const char*     sRootPath );
```

Parameter

Parameter	Description
pContext	Context keeping track of configured settings.
pConnectInfo	Connection information. Passed to the send and receive routines. Typically a socket.
sRootPath	String to root path to prepend such as "/httpdocs". Overrides a root path set with <i>IP_WEBS_ConfigRootPath()</i> on page 499.

Table 20.16: IP_WEBS_ProcessEx() parameter list

Return value

==0: O.K.

!= 0: Error

Additional Information

This function is part of the thread functionality of the Web server. *IP_WEBS_ProcessEx()* is a more flexible version of *IP_WEBS_Process()* and should be used instead. The parameter [pContext](#) is a structure holding all the required function pointers to the routines used to send and receive bytes from/to the client, access the file system, allocate and free memory, etc.

It has to be initialized before usage. Refer to *IP_WEBS_Init()* on page 502 for further information.

Refer to *IP_WEBS_ProcessLastEx()* on page 510 and *IP_WEBS_OnConnectionLimit()* on page 506 for further information about the thread handling of the Web server.

20.11.13 IP_WEBS_ProcessLast()

Description

Processes a HTTP request of a client and closes the connection thereafter.

Prototype

```
int IP_WEBS_ProcessLast (      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 20.17: IP_WEBS_ProcessLast() parameter list

Return value

==0: O.K.

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 818. For details about the parameter [pApplication](#) and the `WEBS_APPLICATION` structure, refer to *Structure WEBS_APPLICATION* on page 549.

Refer to *IP_WEBS_Process()* on page 507 and *IP_WEBS_OnConnectionLimit()* on page 506 for further information.

20.11.14 IP_WEBS_ProcessLastEx()

Description

Processes a HTTP request of a client and closes the connection thereafter.

Prototype

```
int IP_WEBS_ProcessLastEx ( WEBS_CONTEXT *pContext,
                           void          *pConnectInfo
                           const char    *sRootPath );
```

Parameter

Parameter	Description
<code>pContext</code>	Context keeping track of configured settings.
<code>PConntextInfo</code>	Connection information. Passed to the send and receive routines. Typically a socket.
<code>sRootPath</code>	String to root path to prepend such as "/httpdocs". Overrides a root path set with <i>IP_WEBS_ConfigRootPath()</i> on page 499.

Table 20.18: IP_WEBS_ProcessLastEx() parameter list

Return value

==0: O.K.

!= 0: Error

Additional Information

This function is part of the thread functionality of the Web server. This is typically called for the last available connection. *IP_WEBS_ProcessLastEx()* is a more flexible version of *IP_WEBS_Process()* and should be used instead. 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 parameter `pContext` is a structure holding all the required function pointers to the routines used to send and receive bytes from/to the client, access the file system, allocate and free memory, etc.:

It has to be initialized before usage. Refer to *IP_WEBS_Init()* on page 502 for further information.

Refer to *IP_WEBS_ProcessLastEx()* on page 510 and *IP_WEBS_OnConnectionLimit()* on page 506 for further information about the thread handling of the Web server.

20.11.15 IP_WEBS_Redirect()

Description

This routine can send the content of a file from a file system instead of having to send a redirect page first.

Prototype

```
int IP_WEBS_Redirect ( WEBS_OUTPUT *pOutput,
                      const char *sFileName,
                      const char *sMIMEType );
```

Parameter

Parameter	Description
pOutput	[IN] Connection context passed to callback.
sFileName	[IN] Path of file to send.
sMIMEType	[IN] MIME type to use instead of automatically detected MIME type based on file name. Can be NULL.

Table 20.19: IP_WEBS_Redirect() parameter list

Return value

< 0: Error
0: O.K.

Additional information

The function shall only be called if no other data has been sent out before. The page that will be sent is parsed for CGIs the same way as it would be parsed when being directly being accessed by the browser. However the URL accessed by the browser will remain the same and the browser will show the same URL as address.

Example

```
/* *****
 *
 *      _CGI_Redirect
 */
static void _CGI_Redirect(WEBS_OUTPUT *pOutput, const char *sParameters) {
    IP_WEBS_Redirect(pOutput, "/index.htm", NULL); // Redirect back to index
}
```

20.11.16 IP_WEBS_Reset()

Description

This routine resets internal structures of the Web Server.

Prototype

```
void IP_WEBS_Reset ( void );
```

Additional information

As the Web Server is not directly connected to the IP stack itself it can not register to the IP stacks de-initialize process. Once the stack has been de-initialized this routine shall be called before re-initializing the IP stack and using the Web Server again.

20.11.17 IP_WEBS_RetrieveUserContext()

Description

Retrieves a previously stored user context from connection context.

Prototype

```
void * IP_WEBS_RetrieveUserContext ( WEBS_OUTPUT *pOutput );
```

Parameter

Parameter	Description
pOutput	[IN] Connection context passed to callback.

Table 20.20: IP_WEBS_RetrieveUserContext() parameter list

Return value

Previously stored data.

Additional information

A user context retrieved will not reset the stored context. The user stored context remains valid until either set to NULL by the user or the connection being closed.

In case a browser reuses an already opened connection the user context is not reset. This can be used to identify a connection reuse or to exchange data within the same connection. It is user responsibility to make sure that the user context is set back to NULL by the last callback if this behavior is not desired.

20.11.18 IP_WEBS_SendHeader()

Description

Generates and sends a header based on the information passed to this function.

Prototype

```
int 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 20.21: 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 529 or *IP_WEBS_ConfigSendVFileHookHeader()* on page 530. 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.

Depending on the MIME type used the browser may wait for new data forever if the connection is not closed after all data has been transferred. For example "application/octet-stream" will leave the browser waiting forever if transfer size is not sent in the header. Therefore, *IP_WEBS_SendHeader()* on page 514 informs the Web server to close the connection after sending the data.

20.11.19 IP_WEBS_SendHeaderEx()

Description

Generates and sends a header based on the information passed to this function.

Prototype

```
int IP_WEBS_SendHeaderEx ( WEBS_OUTPUT * pContext,
                           const char * sFileName,
                           const char * sMimeType
                           U8      ReqKeepCon );
```

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.
ReqKeepCon	[IN] 1: Keep connection, if possible. 0: Close connection after data transmission.

Table 20.22: IP_WEBS_SendHeader() parameter list

Return value

0: O.K., header sent, Connection will be kept open after transmission.

1: O.K., header sent, Connection will be closed after transmission.

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 529 or *IP_WEBS_ConfigSendVFileHookHeader()* on page 530. 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.

Depending on the MIME type used the browser may wait for new data forever if the connection is not closed after all data has been transferred. For example "application/octet-stream" will leave the browser waiting forever if transfer size is not sent in the header. Therefore, *IP_WEBS_SendHeader()* on page 514 informs the Web server to close the connection after sending the data. In some cases, for example SSE, it is desirable to keep the connection, since it will be used for the following payload transmission. The Web Server maintains the connections and closes them if necessary.

IP_WEBS_SendHeaderEx() informs the Web server, that the connection should kept open if possible.

20.11.20IP_WEBS_SendLocationHeader()

Description

Sends a header with a redirection code for the browser.

Prototype

```
void IP_WEBS_SendLocationHeader (          WEBS_OUTPUT* pOutput,  
                                       const char*      sURI,  
                                       const char*      sCodeDesc );
```

Parameter

Parameter	Description
<code>pOutput</code>	Out context of the connection.
<code>sURI</code>	URI where to redirect the browser.
<code>sCodeDesc</code>	Any code and description that supports the "Location: <URI>" field like: - "301 Moved Permanently" - "302 Found" - "303 See Other"

Table 20.23: IP_WEBS_SendLocationHeader() parameter list

Additional information

A redirect like 303 can be used to forward the user to the same page after POST data has been submitted, resulting in no resending of the POST data upon a refresh of the page.

20.11.21 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 20.24: IP_WEBS_SendMem() parameter list

Return value

0 OK.

20.11.22IP_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 20.25: IP_WEBS_SendString() parameter list

Return value

0 OK.

20.11.23 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
<code>pOutput</code>	[IN] Pointer to the WEBS_OUTPUT structure.
<code>s</code>	[IN] Pointer to a string that should be transmitted.

Table 20.26: 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.

20.11.24IP_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 20.27: IP_WEBS_SendUnsigned() parameter list

Return value

0 OK.

20.11.25 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 20.28: 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 Webpage 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 Webpages, 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 550 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
    }
}
```

20.11.26 IP_WEBS_SetHeaderCacheControl()

Description

Sets the string to be sent for the cache-control field in the header. The field itself has to be part of the string, e.g.: "Cache-Control: no-cache\r\n".

Prototype

```
void IP_WEBS_SetHeaderCacheControl ( const char* sCacheControl );
```

Parameter

Parameter	Description
sCacheControl	Cache control field content.

Table 20.29: IP_WEBS_SetHeaderCacheControl() parameter list

20.11.27 IP_WEBS_StoreUserContext()

Description

Stores a user context into the connection context for using it across several callbacks.

Prototype

```
void IP_WEBS_StoreUserContext ( WEBS_OUTPUT *pOutput,
                                void          *pContext );
```

Parameter

Parameter	Description
pOutput	[IN] Connection context passed to callback.
pContext	[IN] Pointer to store.

Table 20.30: IP_WEBS_StoreUserContext() parameter list

Additional information

Sometimes it might be necessary to exchange information between several callbacks that will be called one after another when a Web page is processed or form data is submitted. The user can use this mechanism to store data into the current connection context in one callback and retrieve the data from another callback of the same connection. Callbacks such as CGIs will be called in the order they are referenced by the Web page. Therefore the order of their accesses is known and can be used in dynamic memory allocation. A sample using pseudo code is shown below.

Examples

```

/*****
*
*      _CGI_1
*
*  Notes
*      This is the first callback accessed for the operation requested
*      by the browser. This is a perfect place to allocate some memory.
*/
static void _CGI_1(WEBS_OUTPUT *pOutput, const char *sParameters, const char *sValue)
{
    char *s;

    s = (char*)OS_malloc(13);                                // Allocate memory for data as
                                                                // data has to remain valid outside
                                                                // of this routine.

    strcpy(s, "Hello world!");                                // Fill with data
    IP_WEBS_StoreUserContext(pOutput, (void*)s);              // Store pointer to text for other
                                                                // callback to access.
}

/*****
*
*      _CGI_2
*
*  Notes
*      This is the last callback accessed for the operation requested
*      by the browser. This is a perfect place to free the previously
*      allocated memory.
*/
static void _CGI_2(WEBS_OUTPUT *pOutput, const char *sParameters, const char *sValue)
{
    char *s;

    s = (char*)IP_WEBS_RetrieveUserContext(pOutput);          // Retrieve previously stored
                                                                // data.
}

```

```
printf("%s", s);
IP_WEBS_StoreUserContext(pOutput, NULL);
OS_free((void*)s);
}
```

```
// Output data.
// Invalidate user context.
// Free allocated memory.
```

20.11.28 IP_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,
                           U8                      ForceEncoding );
```

Parameter

Parameter	Description
pHook	[IN] Pointer to an element of type <code>WEBS_VFILE_HOOK</code> .
pVFileApp	[IN] Pointer to an element of type <code>WEBS_VFILE_APPLICATION</code> .
ForceEncoding	[IN] When set to <code>HTTP_ENCODING_RAW</code> chunked encoding will not be used. Necessary for some implementations such as UPnP.

Table 20.31: 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 552 for detailed information about the structure `WEBS_VFILE_HOOK`. Refer to *Structure WEBS_VFILE_APPLICATION* on page 551 for detailed information about the structure `WEBS_VFILE_APPLICATION`.

Example

```
/* Excerpt from OS_IP_Webserver_UPnP.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 * pContextOut, 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_Webserver_UPnP.c */
//
// UPnP webserver VFile hook
//
static WEBS_VFILE_HOOK _UPnP_VFileHook;

/* Excerpt from OS_IP_Webserver_UPnP.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_Webserver_UPnP.c */

/*****
*
*   MainTask
*/
void MainTask(void);
void MainTask(void) {
    //
    // Activate UPnP with VFile hook for needed XML files
    //
    IP_WEBS_AddVFileHook(&_UPnP_VFileHook, &_UPnP_VFileAPI, HTTP_ENCODING_RAW);
}

```

20.11.29 IP_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
<code>sFilename</code>	[IN] Name of the file.
<code>sExt</code>	[IN] Extension which should be checked.

Table 20.32: IP_WEBS_CompareFilenameExt() parameter list

Return value

0 Match

!= 0 Mismatch

Additional information

The test is case-sensitive, meaning:

<code>IP_WEBS_CompareFilenameExt("Index.html", ".html")</code>	---> Match
<code>IP_WEBS_CompareFilenameExt("Index.htm", ".html")</code>	---> Mismatch
<code>IP_WEBS_CompareFilenameExt("Index.HTML", ".html")</code>	---> Mismatch
<code>IP_WEBS_CompareFilenameExt("Index.html", ".HTML")</code>	---> Mismatch

20.11.30 IP_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 20.33: 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 514. Sending a header has to be done before sending any other content.

20.11.31 IP_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 20.34: 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 514. Sending a header has to be done before sending any other content.

20.11.32 IP_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 20.35: 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 20.36: Example

20.11.33IP_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 20.37: IP_WEBS_DecodeString() parameter list

Return value

0 String does not include url encoded characters. No change.

>0 Length of the decoded string excluding the terminating null character.

20.11.34 IP_WEBS_GetDecodedStrLen()

Description

Returns the length of a HTML encoded string when decoded excluding null character.

Prototype

```
int IP_WEBS_GetDecodedStrLen( const char *s,
                             int    Len );
```

Parameter

Parameter	Description
s	[IN] String.
Len	[IN] Length of input string excluding terminating null character.

Table 20.38: IP_WEBS_GetDecodedStrLen() parameter list

Return value

<0: Error

>0: Length of decoded string excluding terminating null character.

20.11.35IP_WEBS_GetNumParas()

Description

Returns the number of parameter/value pairs.

Prototype

```
int IP_WEBS_GetNumParas ( const char * sParameters );
```

Parameter

Parameter	Description
<code>sParameters</code>	[IN] Zero-terminated string with parameter/value pairs.

Table 20.39: 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.

20.11.36 IP_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 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 value. (Optional, can be NULL.)
ValueLen	[IN] Size of the buffer to store the value. (0 if sValue is NULL.)

Table 20.40: 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`.

20.11.37 IP_WEBS_GetParaValuePtr()

Description

Parses a string for valid parameter/value pairs and returns a pointer to the requested parameter and the length of the parameter 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
<code>sBuffer</code>	[IN] Zero-terminated parameter/value string that should be parsed.
<code>ParaNum</code>	[IN] Zero-based index of the parameter/value pairs.
<code>ppPara</code>	[OUT] Pointer to the pointer locating the start of the requested parameter name. (Optional, can be NULL.)
<code>pParaLen</code>	[OUT] Pointer to a buffer to store the length of the parameter name without termination. (Optional, can be NULL.)
<code>ppValue</code>	[OUT] Pointer to the pointer locating the start of the requested parameter value. (Optional, can be NULL.)
<code>pValueLen</code>	[OUT] Pointer to a buffer to store the length of the parameter value without termination. (Optional, can be NULL.)

Table 20.41: 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 parameters passed by the client without modifying 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\">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&
SEGGER Microcontroller
GmbH & Co. KG &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&
<A HREF=\"http://www.segger.com\">www.segger.com</A></
BODY></HTML>");
}

```

20.11.38 IP_WEBS_GetConnectInfo()

Description

Retrieves the connect info that has been passed to the Web Server process function from the application.

Prototype

```
void* IP_WEBS_GetConnectInfo( WEBS_OUTPUT* pOutput );
```

Parameter

Parameter	Description
pOutput	Out context of the connection.

Table 20.42: IP_WEBS_GetConnectInfo() parameter list

Return value

Connection info passed to process function.

20.11.39 IP_WEBS_GetURI()

Description

Returns the URI of the accessed resource.

Prototype

```
const char* IP_WEBS_GetURI( WEBS_OUTPUT *pOutput,
                           char GetFullURI );
```

Parameter

Parameter	Description
<code>pOutput</code>	[IN] Connection output context.
<code>GetFullURI</code>	[IN] Switch to select between URI and "full URI". URI contains the resource address up to any delimiter such as '?'. The "full URI" contains the complete resource address accessed up to the next whitespace after the resource address including '?' and following characters. 0: URI 1: "full URI"

Table 20.43: IP_WEBS_GetURI() parameter list

Return value

NULL: In case "full URI" has been selected but is not available.
Other: Pointer to URI or "full URI" string.

Additional information

To support storing the "full URI" the define `WEBS_URI_BUFFER_SIZE` needs to be set. If it is not set or its size is too small, requesting the "full URI" will always return NULL.

20.11.40 IP_WEBS_UseRawEncoding()

Description

Overrides the previously selected encoding of the Web Server to use RAW encoding. This also means closing the connection after answering the request.

Prototype

```
void IP_WEBS_UseRawEncoding( WEBS_OUTPUT* pOutput );
```

Parameter

Parameter	Description
pOutput	Out context of the connection.

Table 20.44: IP_WEBS_UseRawEncoding() parameter list

20.11.41 IP_WEBS_METHOD_AddHook()

Description

Registers a callback to serve special content upon call of a METHOD.

Prototype

```
void IP_WEBS_METHOD_AddHook ( WEBS_METHOD_HOOK *pHook,
                              IP_WEBS_pfMethod *pf,
                              const char        *sURI );
```

Parameter

Parameter	Description
pHook	[IN] Pointer to an element of type <code>WEBS_METHOD_HOOK</code> .
pf	[IN] Pointer to a function of type <code>IP_WEBS_pfMethod</code> .
sURI	[IN] URI to listen for requested method.

Table 20.45: IP_WEBS_METHOD_AddHook() parameter list

Additional information

The function can be used to implement Web applications that need to make use of METHODS in a special way such as REST (REpresentational State Transfer) that uses GET and POST in a different way they are typically used by a Web server. Refer to *Structure WEBS_METHOD_HOOK* on page 555 for detailed information about the structure `WEBS_METHOD_HOOK`. Refer to *Callback IP_WEBS_pfMethod* on page 559 for detailed information about the callback parameters of `IP_WEBS_pfMethod`.

Typically one URI on the server is used to serve such a special need and this function allows redefining METHODS for a specific URI for such cases. Locations within this URI such as `/URI/1` in case `/URI` has been defined for the hook are served by the hook as well. In case further hooks are placed inside paths of other hooks the hook with the deepest path matching the requested URI will be used.

Example

```

/* Excerpt from OS_IP_Webserver.c */

/*****
 *
 *      _REST_cb
 *
 * Function description
 *      Callback for demonstrational REST implementation using a METHOD
 *      hook. As there is no clearly defined standard for REST this
 *      implementation shall act as sample and starting point on how
 *      REST support could be implemented by you.
 *
 * Parameters
 *      pContext      - Context for incoming data
 *      pOutput       - Context for outgoing data
 *      sMethod       - String containing used METHOD
 *      sAccept       - NULL or string containing value of "Accept" field of HTTP header
 *      sContentType  - NULL or string containing value of "Content-Type" field of
 *                      HTTP header
 *      sResource     - String containing URI that was accessed
 *      ContentLen    - 0 or length of data submitted by client
 *
 * Return value
 *      0              - O.K.
 *      Other         - Error
 */
static int _REST_cb(      void      *pContext,
                        WEBS_OUTPUT *pOutput,
                        const char  *sMethod,
                        const char  *sAccept,
                        const char  *sContentType,
                        const char  *sResource,
                        U32         ContentLen ) {

    int  Len;
    char acAccept[128];
    char acContentType[32];

    //
    // Strings located at sAccept and sContentType need to be copied to
    // another location before calling any other Web Server API as they
    // will be overwritten.
    //
    if (sAccept) {
        _CopyString(acAccept, sAccept, sizeof(acAccept));
    }
    if (sContentType) {
        _CopyString(acContentType, sContentType, sizeof(acContentType));
    }
    //
    // Send implementation specific header to client
    //
    IP_WEBS_SendHeader(pOutput, NULL, "application/REST");
    //
    // Output information about the METHOD used by the client
    //
    IP_WEBS_SendString(pOutput, "METHOD:      ");
    IP_WEBS_SendString(pOutput, sMethod);
    IP_WEBS_SendString(pOutput, "\n");
    //
    // Output information about which URI has been accessed by the client
    //
    IP_WEBS_SendString(pOutput, "URI:      ");
    IP_WEBS_SendString(pOutput, sResource);
    IP_WEBS_SendString(pOutput, "\n");
    //

```

```

// Output information about "Accept" field given in header sent by client, if any
//
if (sAccept) {
    IP_WEBS_SendString(pOutput, "Accept:          ");
    IP_WEBS_SendString(pOutput, acAccept);
    IP_WEBS_SendString(pOutput, "\n");
}
//
// Output information about "Content-Type" field given in header sent by
// client, if any
//
if (sContentType) {
    IP_WEBS_SendString(pOutput, "Content-Type: ");
    IP_WEBS_SendString(pOutput, acContentType);
}
//
// Output content sent by client, or content previously sent by client that has
// been saved
//
if ((_acRestContent[0] || ContentLen) && sContentType) {
    IP_WEBS_SendString(pOutput, "\n");
}
if (_acRestContent[0] || ContentLen) {
    IP_WEBS_SendString(pOutput, "Content:\n");
}
if (ContentLen) {
    //
    // Update saved content
    //
    Len = SEGGER_MIN(sizeof(_acRestContent) - 1, ContentLen);
    IP_WEBS_METHOD_CopyData(pContext, _acRestContent, Len);
    _acRestContent[ContentLen] = 0;
}
if (_acRestContent[0]) {
    IP_WEBS_SendString(pOutput, _acRestContent);
}
return 0;
}

/*****
*
*      MainTask
*/
void MainTask(void);
void MainTask(void) {
    //
    // Register URI "http://<ip>/REST" for demonstrational REST implementation
    //
    IP_WEBS_METHOD_AddHook(&_MethodHook, _REST_cb, "/REST");
}

```

20.11.42 IP_WEBS_METHOD_CopyData()

Description

Requests incoming data for use in a METHOD callback.

Prototype

```
int IP_WEBS_METHOD_CopyData ( void      *pContext,
                              void      *pBuffer,
                              unsigned   NumBytes );
```

Parameter

Parameter	Description
pContext	[IN] METHOD context for incoming data.
pBuffer	[OUT] Pointer to buffer where incoming data is stored.
NumBytes	[IN] Number of bytes to read.

Table 20.46: IP_WEBS_METHOD_CopyData() parameter list

Return value

<0: Error

0: Connection closed

>0: Number of bytes read

Additional information

The function can be used to implement Web applications that need to make use of METHODS in a special way such as REST (REpresentational State Transfer) that uses GET and POST in a different way they are typically used by a Web server. Refer to *Structure WEBS_METHOD_HOOK* on page 555 for detailed information about the structure `WEBS_METHOD_HOOK`. Refer to *Callback IP_WEBS_pfMethod* on page 559 for detailed information about the callback parameters of `IP_WEBS_pfMethod`.

Typically one URI on the server is used to serve such a special need and this function allows redefining METHODS for a specific URI for such cases. Locations within this URI such as `/URI/1` in case `/URI` has been defined for the hook are served by the hook as well. In case further hooks are placed inside paths of other hooks the hook with the deepest path matching the requested URI will be used.

20.11.43 IP_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
<code>pSrc</code>	[IN] Pointer to data to encode.
<code>SrcLen</code>	Number of bytes to encode.
<code>pDest</code>	[IN] Pointer to the destination buffer.
<code>pDestLen</code>	[IN] Pointer to the destination buffer size. [OUT] Pointer to the number of bytes used in the destination buffer.

Table 20.47: 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>.

20.11.44IP_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
<code>pSrc</code>	[IN] Pointer to data to encode.
<code>SrcLen</code>	Number of bytes to encode.
<code>pDest</code>	[IN] Pointer to the destination buffer.
<code>pDestLen</code>	[IN] Pointer to the destination buffer size. [OUT] Pointer to the number of bytes used in the destination buffer.

Table 20.48: 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>.

20.12 Web server data structures

20.12.1 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 20.49: Structure WEBS_CGI member list

Additional information

Refer to *Common Gateway Interface (CGI)* on page 470 for detailed information about the use of this structure.

20.12.2 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
<code>sPath</code>	A string which defines the path of the resources.
<code>sRealm</code>	A string which defines the realm which requires authentication. Optional, can be NULL.
<code>sUserPass</code>	A string containing the user name/password combination. Optional, can be NULL.

Table 20.50: 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 480 for detailed information about the HTTP Basic Authentication scheme.

20.12.3 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 20.51: Structure WEBS_APPLICATION member list

20.12.4 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 validity.
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 20.52: Structure IP_WEBS_FILE_INFO member list

20.12.5 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 20.53: Structure WEBS_VFILE_APPLICATION member list

20.12.6 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 20.54: Structure WEBS_VFILE_HOOK member list

Additional information

Refer to *Structure WEBS_VFILE_HOOK* on page 552 for detailed information about the structure WEBS_VFILE_HOOK. Refer to *Structure WEBS_VFILE_APPLICATION* on page 551 for detailed information about the structure WEBS_VFILE_APPLICATION.

20.12.7 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 20.55: Structure WEBS_FILE_TYPE member list

20.12.8 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 20.56: Structure WEBS_VFILE_HOOK member list

Additional information

Refer to *Structure WEBS_FILE_TYPE_HOOK* on page 554 for detailed information about the structure WEBS_FILE_TYPE_HOOK. Refer to *Structure WEBS_FILE_TYPE* on page 553 for detailed information about the structure WEBS_FILE_TYPE.

20.12.9 Structure WEBS_METHOD_HOOK

Description

Used to extend the usage of METHODS in the Web server for a given URI.

Prototype

```
typedef struct WEBS_METHOD_HOOK {
    struct WEBS_METHOD_HOOK *pNext;
    IP_WEBS_pfMethod pf;
    const char *sURI;
} WEBS_FILE_TYPE_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_METHOD_HOOK.
pf	Pointer to callback handling the requested method of type Callback IP_WEBS_pfMethod .
sURI	URI registered for METHODS callback.

Table 20.57: Structure WEBS_METHOD_HOOK member list

20.12.10Structure WEBS_PRE_CONTENT_OUTPUT_HOOK

Description

Used to intercept the Web server before content is generated and sent.

Prototype

```
typedef struct WEBS_PRE_CONTENT_OUTPUT_HOOK {
    struct WEBS_PRE_CONTENT_OUTPUT_HOOK* pNext;
    IP_WEBS_pfPreContentOutput    pf;
} WEBS_PRE_CONTENT_OUTPUT_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_PRE_CONTENT_OUTPUT_HOOK.
pf	Pointer to callback of type Callback IP_WEBS_pfPreContentOutput.

Table 20.58: Structure WEBS_PRE_CONTENT_OUTPUT_HOOK member list

20.12.11 Structure WEBS_REQUEST_NOTIFY_HOOK

Description

Used to get notified of incoming requests to the Web server.

Prototype

```
typedef struct WEBS_REQUEST_NOTIFY_HOOK {
    struct WEBS_REQUEST_NOTIFY_HOOK *pNext;
    IP_WEBS_pfRequestNotify    pf;
} WEBS_REQUEST_NOTIFY_HOOK;
```

Member	Description
pNext	Pointer to the previously registered element of WEBS_REQUEST_NOTIFY_HOOK.
pf	Pointer to callback handling the requested method of type Callback IP_WEBS_pfRequestNotify .

Table 20.59: Structure WEBS_REQUEST_NOTIFY_HOOK member list

20.12.12Structure WEBS_REQUEST_NOTIFY_INFO

Description

Used to pass information to a callback registered to be notified upon a request to the Web server.

Prototype

```
typedef struct WEBS_REQUEST_NOTIFY_INFO {  
    const char *sUri;  
    U8      Method;  
} WEBS_REQUEST_NOTIFY_INFO;
```

Member	Description
sUri	Pointer to string containing the requested location.
Method	HTTP METHOD used in the request: - METHOD_GET - METHOD_HEAD - METHOD_POST

Table 20.60: Structure WEBS_REQUEST_NOTIFY_INFO member list

20.12.13 Callback IP_WEBS_pfMethod

Description

Used to extend the usage of METHODS in the Web server for a given URI.

Prototype

```
typedef int (*IP_WEBS_pfMethod) (          void      *pContext,
                                     WEBS_OUTPUT *pOutput,
                                     const char   *sMethod,
                                     const char   *sAccept,
                                     const char   *sContentType,
                                     const char   *sResource,
                                     U32         ContentLen );
```

Member	Description
pContext	[IN] METHOD context for incoming data used with IP_WEBS_METHOD_* routines.
pOutput	[IN] Output context for IP_WEBS_* routines.
sMethod	[IN] String containing METHOD requested by client.
sAccept	[IN] String containing value of "Accept" field of header sent by client. May be NULL in case there was no such field.
sContentType	[IN] String containing value of "Content-Type" field of header sent by client. May be NULL in case there was no such field.
sResource	String containing URI that was accessed.
ContentLen	Length of data submitted by client that can be read. 0 in case no data was sent by client.

Table 20.61: Callback IP_WEBS_pfMethod parameter list

Warning: Strings located at sAccept and sContentType need to be copied to another location before calling any other Web Server API as they will be overwritten.

20.12.14 Callback IP_WEBS_pfPreContentOutput

Description

Called before content is generated and sent by the Web server in regular cases like VFiles and files from a filesystem.

Prototype

```
typedef void (*IP_WEBS_pfPreContentOutput)( WEBS_OUTPUT* pOutput );
```

Member	Description
pOutput	Output context for IP_WEBS_* routines.

Table 20.62: Callback IP_WEBS_pfPreContentOutput parameter list

20.12.15 Callback IP_WEBS_pfRequestNotify

Description

Called upon a request to the Web server.

Prototype

```
typedef void (*IP_WEBS_pfRequestNotify) ( WEBS_REQUEST_NOTIFY_INFO* pInfo );
```

Member	Description
<code>pInfo</code>	Pointer to structure containing information about the current request being handled of type <code>Structure WEBS_REQUEST_NOTIFY_INFO</code> .

Table 20.63: Callback IP_WEBS_pfRequestNotify parameter list

Warning: Data located in pInfo needs to be copied to another location if they shall be used outside of the callback as they will be overwritten.

20.13 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       512
#define WEBS_PARA_BUFFER_SIZE       256
#define WEBS_ERR_BUFFER_SIZE        128
#define WEBS_AUTH_BUFFER_SIZE       32
#define WEBS_FILENAME_BUFFER_SIZE   32
#define WEBS_UPLOAD_FILENAME_BUFFER_SIZE 64
```

20.13.1 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 7.0Kbyte

Table 20.64: Web server ROM usage ARM7

20.13.2 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 6.5Kbyte

Table 20.65: Web server ROM usage Cortex-M3

20.13.3 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 1000 bytes
ChildTask	Handles a request.	approximately 3000 bytes

Table 20.66: 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 * 3 kBytes)

Example: Web server accepting only 1 connection

RAM usage = 0.1 kBytes + 1000 +(1 * 3 kBytes)

RAM usage = 4.1 kBytes

Example: Web server accepting up to 3 connections in parallel

RAM usage = 0.1 kBytes + 1000 + (3 * 3 kBytes)

RAM usage = 10.1 kBytes

Chapter 21

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.

21.1 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]	<i>SMTP Service Extension for Authentication</i> 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 <i>Configuration</i> on page 569 for detailed information.
Inc	Contains the required include files.
IP	Contains the SMTP client sources, <code>IP_SMTPC.c</code> and <code>IP_SMTPC.h</code> .
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

21.2 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.

21.3 Requirements

TCP/IP stack

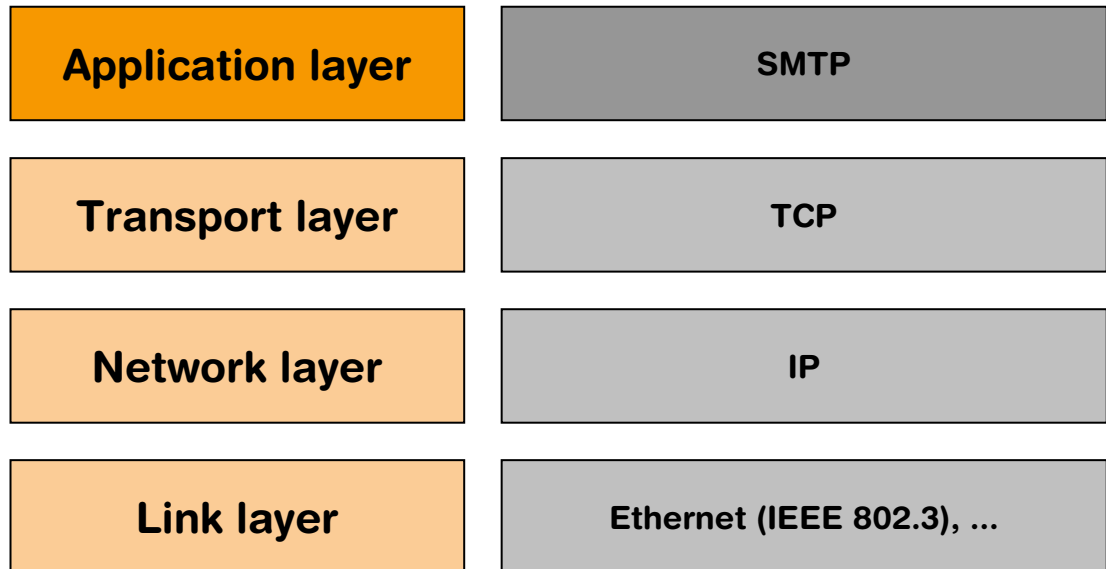
The embOS/IP SMTP client requires a TCP/IP stack. It is optimized for embOS/IP, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of embOS/IP.

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.

21.4 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 LOGIN 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
```


21.5 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).

21.5.1 Compile time configuration switches

Type	Symbolic name	Default	Description
F	<code>SMTPC_WARN</code>	--	Defines a function to output warnings. In debug configurations (<code>DEBUG == 1</code>) <code>SMTPC_WARN</code> maps to <code>IP_Warnf_Application()</code> .
F	<code>SMTPC_LOG</code>	--	Defines a function to output logging messages. In debug configurations (<code>DEBUG == 1</code>) <code>SMTPC_LOG</code> maps to <code>IP_Logf_Application()</code> .
N	<code>SMTPC_SERVER_PORT</code>	25	Defines the port where the SMTP server is listening.
N	<code>SMTPC_IN_BUFFER_SIZE</code>	256	Defines the size of the input buffer. The input buffer is used to store the SMTP replies of the SMTP server.
N	<code>SMTPC_AUTH_USER_BUFFER_SIZE</code>	48	Defines the size of the buffer used for the Base-64 encoded user name.
N	<code>SMTPC_AUTH_PASS_BUFFER_SIZE</code>	48	Defines the size of the buffer used for the Base-64 encoded password.

21.6 API functions

Function	Description
SMTP client functions	
IP_SMTPC_Send()	Sends an email to a mail transfer agent.

Table 21.1: embOS/IP SMTP client interface function overview

21.6.1 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_SMTPC_API structure. Refer to <i>Structure IP_SMTPC_API</i> on page 573 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 <i>Structure IP_SMTPC_MAIL_ADDR</i> on page 576 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 <i>Structure IP_SMTPC_MESSAGE</i> on page 578 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 <i>Structure IP_SMTPC_MTA</i> on page 579 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 <i>Structure IP_SMTPC_APPLICATION</i> on page 575 for detailed information about the elements of the IP_SMTPC_APPLICATION structure.

Table 21.2: IP_SMTPC_Send() parameter list

Return value

- 0 OK.
- 1 Error.

21.7 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 21.3: embOS/IP SMTP client interface function overview

21.7.1 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, <code>connect()</code>).
pfDisconnect	Pointer to the disconnect function (for example, <code>closesocket()</code>).
pfSend	Pointer to a callback function (for example, <code>send()</code>).
pfDisconnect	Pointer to a callback function (for example, <code>recv()</code>).

Table 21.4: 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(("Socket 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
};

```

21.7.2 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
    const char * sTimezone;  // Time zone.
} 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, <code>sample.com</code> . According to RFC 821 the maximum total length of a domain name or number is 64 characters.
sTimezone	Time zone. The zone specifies the offset from Coordinated Universal Time (UTC). Offset from UTC is passed as string: <code>" +0100"</code> . Can be NULL.

Table 21.5: 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.
};
```

21.7.3 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 21.6: 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 571.

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);
}

```

21.7.4 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 21.7: Structure IP_SMTPC_MESSAGE member list

21.7.5 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 <code>NULL</code> .
sPass	Account password (for example, "password"). Can be <code>NULL</code> .

Table 21.8: 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.

21.8 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
```

21.8.1 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.

21.8.1.1 ROM usage

Addon	ROM
embOS/IP SMTP client	approximately 7.1Kbyte

Table 21.9: SMTPC client ROM usage ARM7

21.8.1.2 RAM usage

Addon	RAM
embOS/IP SMTP client (w/o task stack)	approximately 4.7Kbyte

Table 21.10: SMTPC client RAM usage ARM7

21.8.2 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.

21.8.2.1 ROM usage

Addon	ROM
embOS/IP SMTP client	approximately 6.5Kbyte

Table 21.11: SMTPC client ROM usage Cortex-M3

21.8.2.2 RAM usage

Addon	RAM
embOS/IP SMTP client w/o task stack	approximately 4.7Kbyte

Table 21.12: SMTPC client RAM usage Cortex-M3

Chapter 22

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.

22.1 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 <i>File system abstraction layer function table</i> on page 819 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

22.2 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.

22.3 Requirements

TCP/IP stack

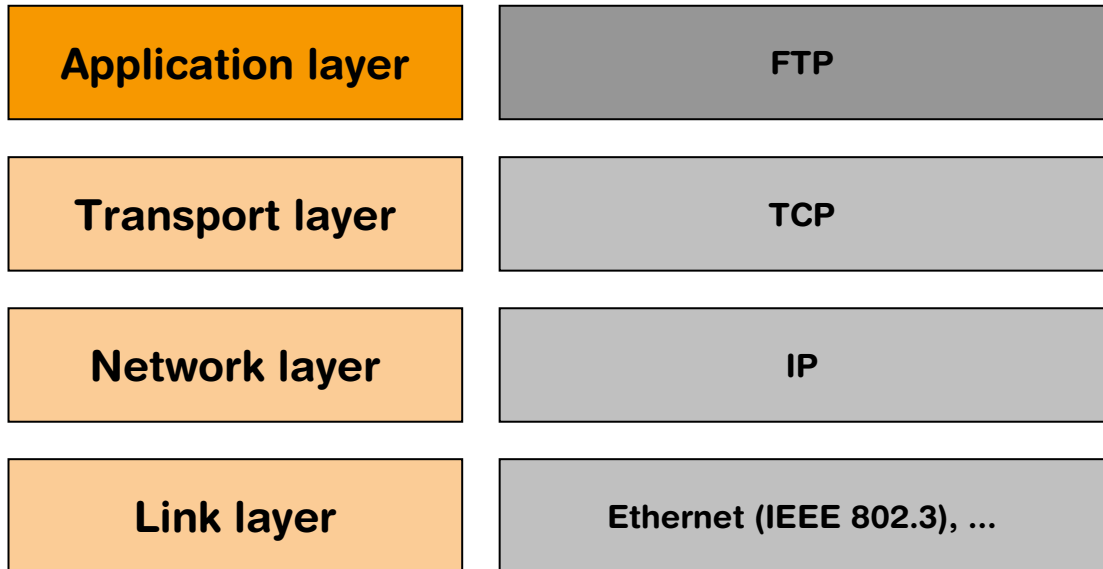
The embOS/IP FTP server requires a TCP/IP stack. It is optimized for embOS/IP, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of embOS/IP.

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.

22.4 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.



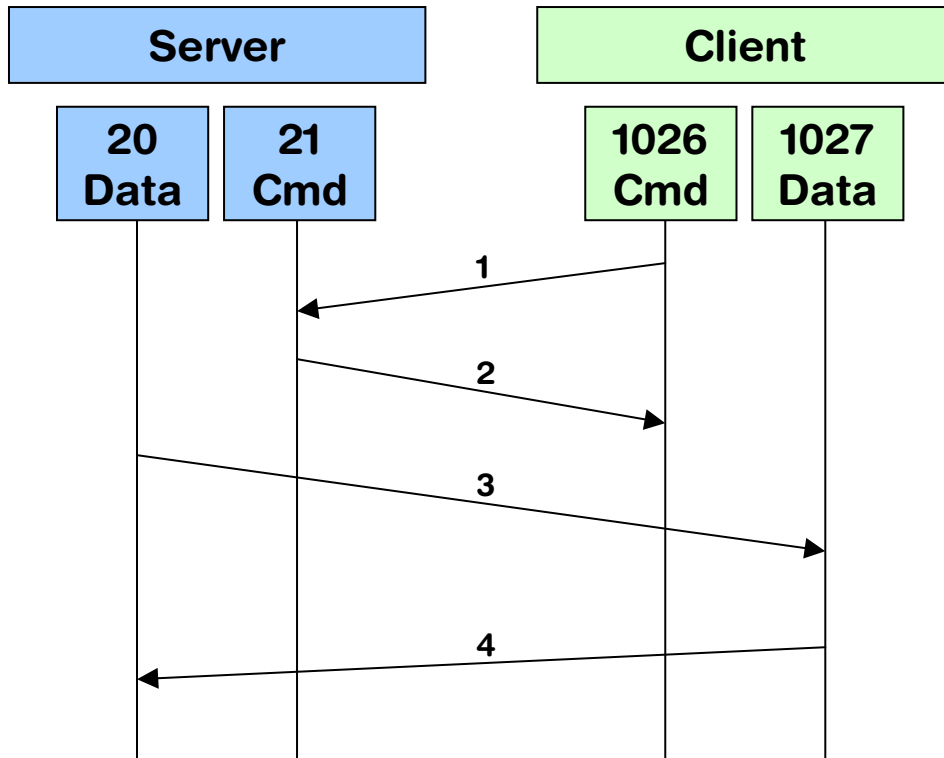
When an FTP client contacts a server, a TCP connection is established between the two machines. The server does a passive open (a socket is listen) when it begins operation; thereafter clients can connect with the server via active opens. This TCP connection persists for as long as the client maintains 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.

22.4.1 Active mode FTP

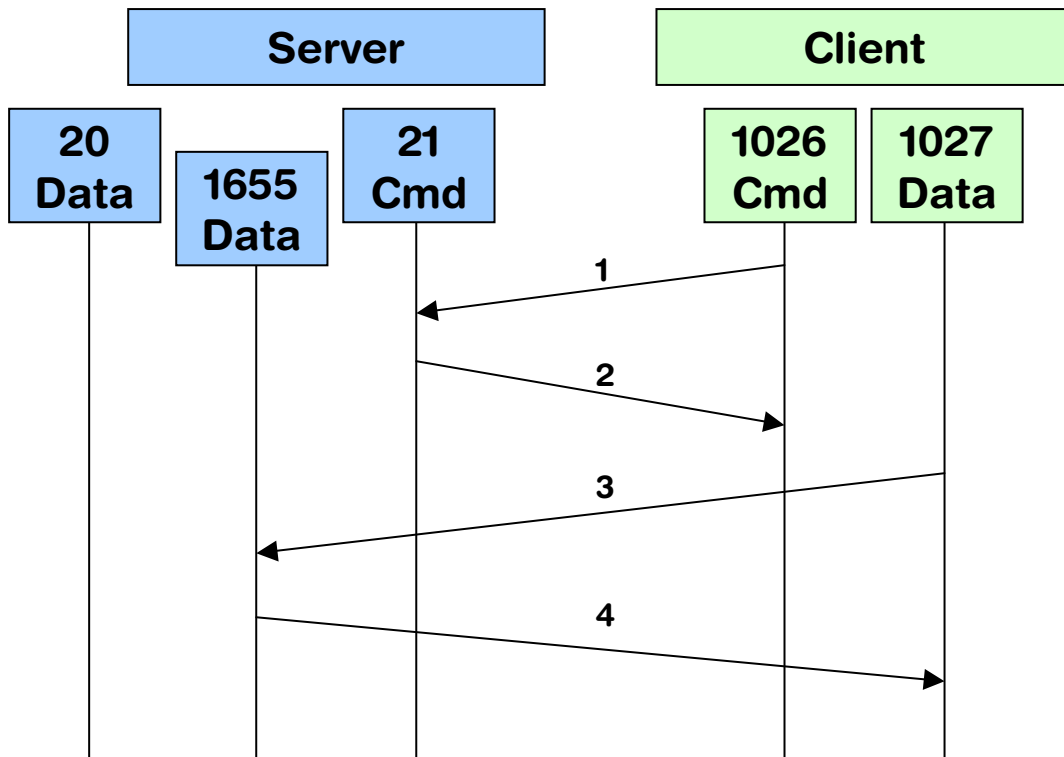
In active mode FTP the client connects from a random unprivileged port P ($P > 1023$) to the FTP server's command port, port 21. Then, the client starts listening to port $P+1$ and sends the FTP command `PORT P+1` to the FTP server. The server will then connect back to the client's specified data port from its local data port, which is port 20.



22.4.2 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.



22.4.3 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.

22.4.4 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 22.1: embOS/IP FTP commands

22.5 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 818 for detailed information.

22.5.1 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, a 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".

22.5.2 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.

22.6 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 to 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 604.

22.6.1 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
<code>sUser</code>	[IN] User name.

Table 22.2: 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;
}
```

22.6.2 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 22.3: 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;
    }
}
```


22.6.3 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 22.4: 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;
}
```

22.6.4 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 22.5: pfGetFileInfo() parameter list

Additional information

Providing a function for file permissions is optional. If using permissions 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;
}
```

22.7 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).

22.7.1 Compile time configuration switches

Type	Symbolic name	Default	Description
F	<code>FTPS_WARN</code>	--	Defines a function to output warnings. In debug configurations (<code>DEBUG == 1</code>) <code>FTPS_WARN</code> maps to <code>IP_Warnf_Application()</code>
F	<code>FTPS_LOG</code>	--	Defines a function to output logging messages. In debug configurations (<code>DEBUG == 1</code>) <code>FTPS_LOG</code> maps to <code>IP_Logf_Application()</code> .
N	<code>FTPS_AUTH_BUFFER_SIZE</code>	32	Defines the size of the buffer used for the authentication string.
N	<code>FTPS_BUFFER_SIZE</code>	512	Defines the size of the send and receive buffer of the FTP server.
N	<code>FTPS_MAX_PATH</code>	128	Defines the maximum length of the buffer used for the path string.
N	<code>FTPS_MAX_PATH_DIR</code>	64	Defines the maximum length of the buffer used for the directory string.
N	<code>FTPS_ERR_BUFFER_SIZE</code>	256	Defines the size of the buffer used for the authentication string.

22.7.2 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 sent. 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:

```
-rw-r--r-- 1 root 5072 Jan 1 1980 PAKET00.TXT
```

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:

```
-rw-r--r-- 1 root 5070 Jul 29 11:04 PAKET01.TXT
```

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 599 for detailed information.

22.7.2.1 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 since 1980 (0-127)

This function pointer is used in the `FTPS_APPLICATION` structure. Refer to *Structure FTPS_APPLICATION* on page 605 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;
}
```

22.8 API functions

Function	Description
<code>IP_FTPS_Process()</code>	Initializes and starts the embOS/IP FTP server.
<code>IP_FTPS_OnConnectionLimit()</code>	Returns when the connection is closed or a fatal error occurs.

Table 22.6: embOS/IP FTP server interface function overview

22.8.1 IP_FTPS_Process()

Description

Initializes and starts the FTP server.

Prototype

```
int IP_FTPS_Process ( const IP_FTPS_API      * pIP_API,
                     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 22.7: 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 817. For detailed information about the structure type FTPS_APPLICATION refer to *Structure FTPS_APPLICATION* on page 605.

22.8.2 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 22.8: 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.

22.9 FTP server data structures

22.9.1 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 22.9: Structure IP_FTPS_API member list

22.9.2 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 permisisions of the connected user for every file. May be NULL if directory permisisions are sufficient for your needs.

Table 22.10: Structure FTPS_ACCESS_CONTROL member list

Example

Refer to *Access control* on page 591 for an example.

22.9.3 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_APPLICATION structure.
pfGetTimeDate	Pointer to the function which returns the current system time.

Table 22.11: Structure FTPS_APPLICATION member list

Example

For additional information to structure FTPS_ACCESS_APPLICATION refer to *Structure FTPS_ACCESS_CONTROL* on page 604. For additional information to function pointer `pfGetTimeDate()` refer to *FTP server system time* on page 598.

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
};
```

22.10 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 Cortex-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)
```

22.10.1 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 22.12: FTP server ROM usage ARM7

22.10.2 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 22.13: FTP server ROM usage Cortex-M3

22.10.3 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 22.14: 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 23

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.

23.1 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 <i>File system abstraction layer function table</i> on page 819 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

23.2 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.

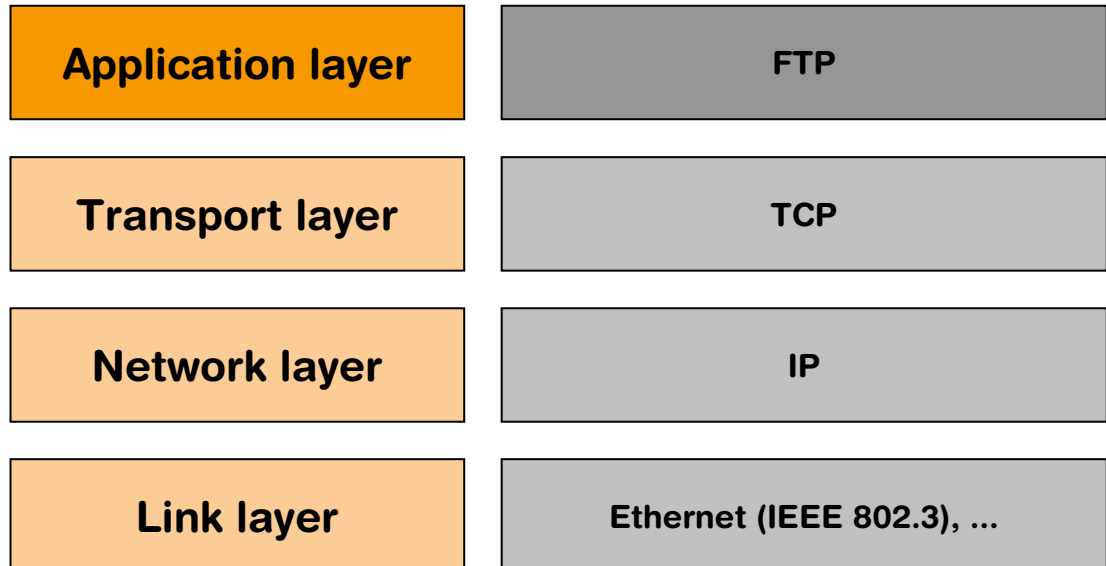
23.3 Requirements

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.

23.4 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.

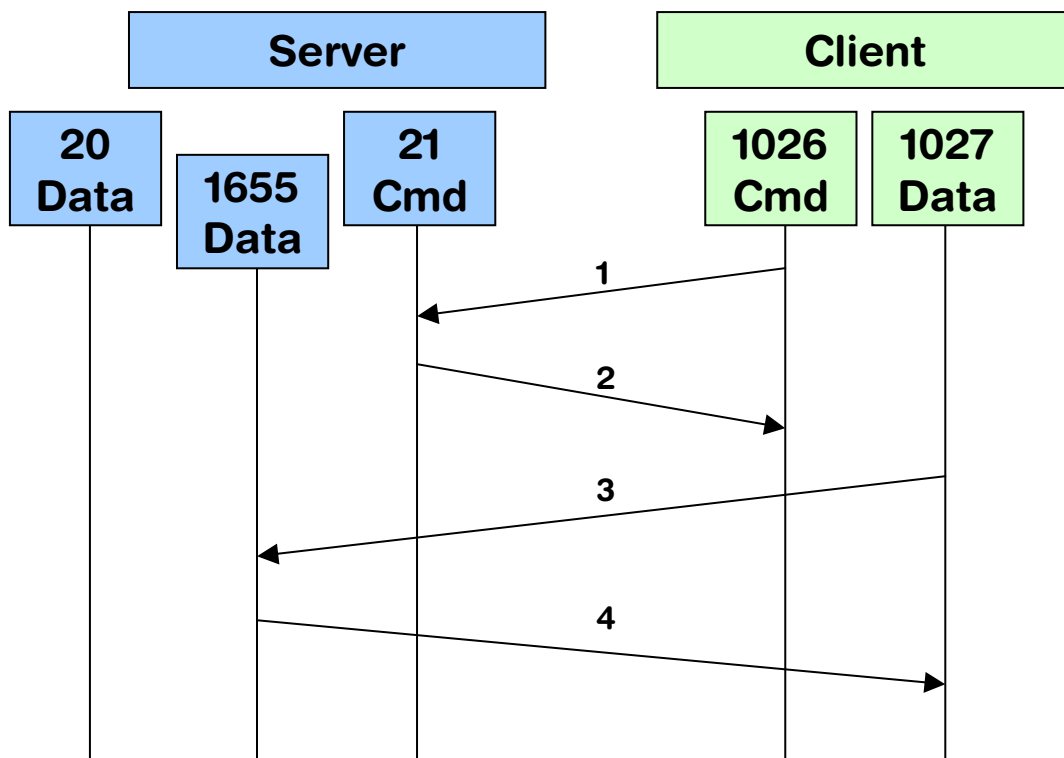
The FTP commands from the client to the server consist of short sets of ASCII characters, followed by optional command parameters. For example, the FTP command to display the current working directory is `PWD` (Print Working Directory). All commands are terminated by a carriage return-linefeed sequence (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.

23.4.1 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.

23.4.2 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 23.1: embOS/IP FTP client commands

23.5 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).

23.5.1 Compile time configuration switches

Type	Symbolic name	Default	Description
F	<code>FTPC_WARN</code>	--	Defines a function to output warnings. In debug configurations (<code>DEBUG == 1</code>) <code>FTPC_WARN</code> should be mapped to <code>IP_Warnf_Application()</code>
F	<code>FTPC_LOG</code>	--	Defines a function to output logging messages. In debug configurations (<code>DEBUG == 1</code>) <code>FTPC_LOG</code> should be mapped to <code>IP_Logf_Application()</code> .
N	<code>FTPC_BUFFER_SIZE</code>	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, <code>FTPC_BUFFER_SIZE == 512</code> means 1024 bytes RAM requirement.
N	<code>FTPC_CTRL_BUFFER_SIZE</code>	256	Defines the maximum length of the buffer used for the control channel.
N	<code>FTPC_SERVER_REPLY_BUFFER_SIZE</code>	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.

23.6 API functions

Function	Description
<code>IP_FTPC_Connect()</code>	Establishes a connection to a FTP server.
<code>IP_FTPC_Disconnect()</code>	Closes an established connection to a FTP server.
<code>IP_FTPC_ExecCmd()</code>	Sends a command to a FTP server.
<code>IP_FTPC_Init()</code>	Initializes the embOS/IP FTP client.

Table 23.2: embOS/IP FTP client interface function overview

23.6.1 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
<code>pContext</code>	[IN] Pointer to a structure of type <code>IP_FTPC_CONTEXT</code> .
<code>sServer</code>	[IN] Dot-decimal IP address of a FTP server, for example "192.168.1.55".
<code>sUser</code>	[IN] User name if required for the authentication. Can be <code>NULL</code> .
<code>sPass</code>	[IN] Password if required for the authentication. Can be <code>NULL</code> .
<code>PortCmd</code>	[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.
<code>Mode</code>	[IN] FTP transfer mode.

Table 23.3: IP_FTPC_Connect() parameter list

Valid values for parameter Mode	Description
<code>FTPC_MODE_PASSIVE</code>	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 622.

Note: In the current version of embOS/IP, the FTP client supports only passive mode FTP.

Example

Refer to *IP_FTPC_ExecCmd()* on page 618 for an example application which uses `IP_FTPC_Connect()`.

23.6.2 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 <code>IP_FTPC_CONTEXT</code> .

Table 23.4: 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 618 for an example application which uses `IP_FTPC_Disconnect()`.

23.6.3 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
<code>pContext</code>	[IN] Pointer to a structure of type <code>IP_FTPC_CONTEXT</code> .
<code>Cmd</code>	[IN] See table below.
<code>sPara</code>	[IN] String with the required parameters for the command. Depending on the command, the parameter can be <code>NULL</code> .

Table 23.5: IP_FTPC_ExecCmd() parameter list

Valid values for parameter <code>Cmd</code>	Description
<code>FTPC_CMD_CDUP</code>	The command CDUP (Change to Parent Directory). <code>sPara</code> is <code>NULL</code> .
<code>FTPC_CMD_CWD</code>	The command CWD (Change Working Directory). <code>sPara</code> is the path to the directory that should be accessed.
<code>FTPC_CMD_LIST</code>	The command LIST (List current directory content). <code>sPara</code> is <code>NULL</code> .
<code>FTPC_CMD_MKD</code>	The command MKD (Make directory). <code>sPara</code> is the name of the directory that should be created.
<code>FTPC_CMD_PASS</code>	The command PASS (Set password). <code>sPara</code> is the password.
<code>FTPC_CMD_PWD</code>	The command PWD (Print Working Directory). <code>sPara</code> is <code>NULL</code> .
<code>FTPC_CMD_RETR</code>	The command RETR (Retrieve). <code>sPara</code> 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.
<code>FTPC_CMD_RMD</code>	The command RMD (Remove directory). <code>sPara</code> is the name of the directory that should be removed.
<code>FTPC_CMD_STOR</code>	The command STOR (Store). <code>sPara</code> 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.
<code>FTPC_CMD_TYPE</code>	The command TYPE (Transfer type). <code>sPara</code> is the transfer type.
<code>FTPC_CMD_USER</code>	The command USER (Set username). <code>sPara</code> is the username.
<code>FTPC_CMD_DELE</code>	The command DELE (delete file). <code>sPara</code> is the name of the file to delete.

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 622 for detailed information about how to initialize the FTP client and refer to *IP_FTPC_Connect()* on page 616 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 613, 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 required.

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(&FTPCConnection, 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(&FTPCConnection, 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(&FTPCConnection, 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(&FTPCConnection);
    FTPC_LOG(("APP: Done.\r\n"));
    BSP_ClrLED(0);
    OS_Delay (10000);
}
}

```

23.6.4 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 <code>IP_FTPC_CONTEXT</code> .
pIP_API	[IN] Pointer to a structure of type <code>IP_FTPC_API</code> .
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 23.6: 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 817. For detailed information about the structure type `IP_FTPC_API` refer to *Appendix A - File system abstraction layer* on page 817.

Example

Refer to `IP_FTPC_ExecCmd()` on page 618 for an example application which uses `IP_FTPC_Init()`.

23.7 FTP client data structures

23.7.1 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 23.7: Structure IP_FTPC_API member list

23.8 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.
```

23.8.1 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 23.8: FTP client ROM usage ARM7

23.8.2 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 23.9: FTP client ROM usage Cortex-M3

23.8.3 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 23.10: FTP client RAM usage release build

The approximately task stack size required for the FTP client can be calculated as follows:

$$\text{TaskStackSize} = 2 * \text{FTPC_BUFFER_SIZE} + \text{FTPC_CTRL_BUFFER_SIZE}$$

Build	Description	RAM
Debug	A task used for the FTP client with debugging features and enabled debug outputs.	app. 1550 bytes

Table 23.11: FTP client RAM usage debug build

The approximately task stack size required for the FTP client can be calculated as follows:

$$\begin{aligned} \text{TaskStackSize} = & 2 * \text{FTPC_BUFFER_SIZE} + \text{FTPC_CTRL_BUFFER_SIZE} \\ & + \text{FTPC_SERVER_REPLY_BUFFER_SIZE} \end{aligned}$$

Chapter 24

TFTP client/server

The TFTP (Trivial File Transfer Protocol) is an extension to the TCP/IP stack. All functions which are required to add a TFTP client or a TFTP server to your application are described in this chapter.

24.1 embOS/IP TFTP

The embOS/IP TFTP is an extension which adds the TFTP protocol to the stack. TFTP stands for Trivial File Transfer Protocol. It is the basic mechanism for moving files via UDP between machines over IP based networks. TFTP 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.

The TFTP server implements the relevant parts of the following RFCs.

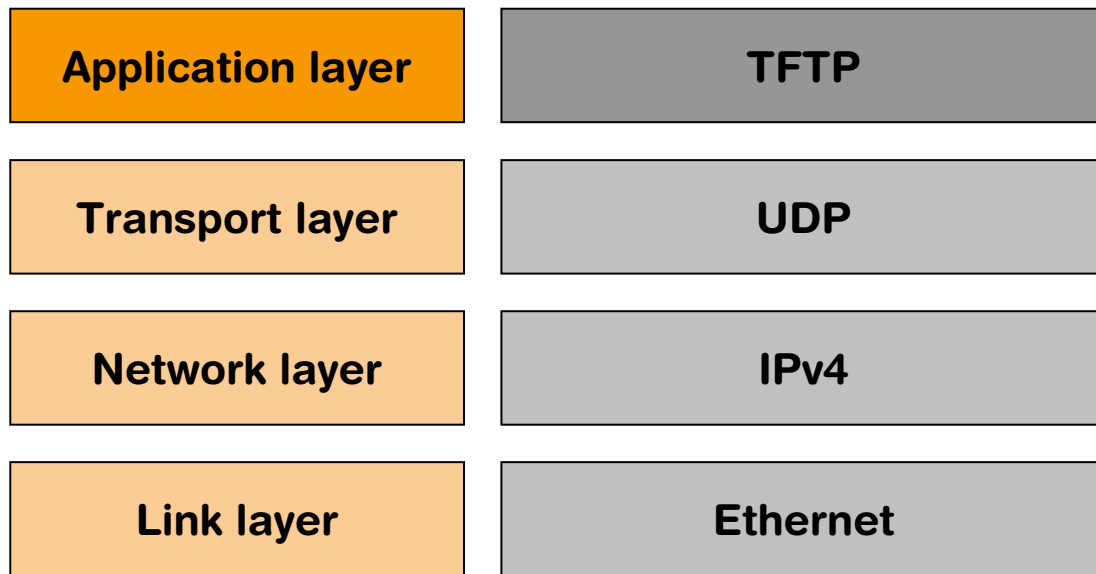
RFC#	Description
[RFC 1350]	TFTP - THE TFTP PROTOCOL (REVISION 2) Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc1350.txt

24.2 Feature list

- Low memory footprint.
- 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.

24.3 TFTP basics

The Trivial File Transfer Protocol (TFTP) is an application layer protocol.



When a TFTP client contacts a server, a UDP command is sent to the servers port. The traditional port is 69. The command sent is either a read or a write request. The client will send data always to the servers port whereas the server will respond with data to the port on that the client is sending.

The TFTP requests are sent in a RFC conform format.

24.4 Using the TFTP samples

Ready to use examples for embOS/IP are supplied. The sample applications are configured to work with each other but can be used with any TFTP client/server with minimal modification. The example applications requires a file system to make data files available. Refer to *File system abstraction layer* on page 818 for detailed information.

24.4.1 Running the TFTP server example on target hardware

The embOS/IP TFTP sample applications should always be the first step to check the proper function of the TFTP client/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 applications can be used on the most targets without the need for changing any of the configuration flags.

24.5 API functions

Function	Description
<code>IP_TFTP_InitContext()</code>	Initialize the TFTP context.
<code>IP_TFTP_RecvFile()</code>	Receive a file from a server.
<code>IP_TFTP_SendFile()</code>	Send a file to a server.
<code>IP_TFTP_ServerTask()</code>	Start the TFTP server.

Table 24.1: embOS/IP TFTP client/server function overview

24.5.1 IP_TFTP_InitContext()

Description

Initializes the context for storing connection parameters of a TFTP client/server.

Prototype

```
int IP_TFTP_InitContext (      TFTP_CONTEXT *pContext,
                              unsigned      IFace,
                              const IP_FS_API *pFS_API,
                              char          *pBuffer,
                              int           BufferSize,
                              U16          ServerPort );
```

Parameter

Parameter	Description
pContext	[IN] Pointer to a structure of type TFTP_CONTEXT.
IFace	[IN] Zero-based interface index to use for UDP transfer.
pFS_API	[IN] Pointer to the used file system API.
pBuffer	[IN] Pointer to buffer for storing transfer data. Needs to be big enough to hold the biggest TFTP message (512 bytes payload + 4 bytes TFTP header).
BufferSize	[IN] Size of buffer assigned with pBuffer .
ServerPort	Server port to open if server.

Table 24.2: IP_TFTP_InitContext() parameter list

Return value

0: O.K.
 < 0: Error, typically buffer too small or no buffer set.

Additional information

A static structure of TFTP_CONTEXT needs to be supplied by the application to supply space to store connection parameters.

24.5.2 IP_TFTP_RecvFile()

Description

Requests a file from a TFTP server.

Prototype

```
int IP_TFTP_RecvFile (      TFTP_CONTEXT *pContext,
                           unsigned      IFace,
                           U32           IPAddr,
                           U16           Port,
                           const char    *sFileName,
                           int           Mode );
```

Parameter

Parameter	Description
pContext	[IN] Pointer to a structure of type TFTP_CONTEXT.
IFace	[IN] Zero-based interface index to use for UDP transfer.
IPAddr	[IN] IP addr. of TFTP server.
Port	[IN] Port of TFTP server listening.
sFileName	[IN] Name of file to retrieve from server.
Mode	[IN] TFTP_MODE_OCTET.

Table 24.3: IP_TFTP_RecvFile() parameter list

Return value

SOCKET_ERROR (-1): Error.
Other: O.K.

Additional information

A static structure of TFTP_CONTEXT needs to be initialized with *IP_TFTP_InitContext()* on page 631 before using it with this function.

24.5.3 IP_TFTP_SendFile()

Description

Sends a file to a TFTP server.

Prototype

```
int IP_TFTP_SendFile (      TFTP_CONTEXT *pContext,
                           unsigned      IFace,
                           U32          IPAddr,
                           U16          Port,
                           const char    *sFileName,
                           int           Mode );
```

Parameter

Parameter	Description
pContext	[IN] Pointer to a structure of type TFTP_CONTEXT.
IFace	[IN] Zero-based interface index to use for UDP transfer.
IPAddr	[IN] IP addr. of TFTP server.
Port	[IN] Port of TFTP server listening.
sFileName	[IN] Name of file to send to server.
Mode	[IN] TFTP_MODE_OCTET.

Table 24.4: IP_TFTP_SendFile() parameter list

Return value

SOCKET_ERROR (-1): Error.

Other: O.K.

Additional information

A static structure of TFTP_CONTEXT needs to be initialized with *IP_TFTP_InitContext()* on page 631 before using it with this function.

24.5.4 IP_TFTP_ServerTask()

Description

TFTP server task that can be started in a separate task.

Prototype

```
void IP_TFTP_ServerTask ( void *pPara );
```

Parameter

Parameter	Description
pPara	[IN] Casted pointer to a structure of type <code>TFTP_CONTEXT</code> .

Table 24.5: IP_TFTP_RecvFile() parameter list

Additional information

A static structure of `TFTP_CONTEXT` needs to be initialized with *IP_TFTP_InitContext()* on page 631 before using it with this function. The task does not return.

24.6 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the TFTP client/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.

24.6.1 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 TFTP client	approximately 1.2Kbyte
embOS/IP TFTP server	approximately 1.2Kbyte

Table 24.6: TFTP client/server ROM usage ARM7

24.6.2 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 FTP client	approximately 1.2Kbyte
embOS/IP FTP server	approximately 1.2Kbyte

Table 24.7: TFTP client/server ROM usage Cortex-M3

24.6.3 RAM usage:

Each connection requires approximately 550 bytes of RAM that split into space for the required transfer buffer (app. 516 bytes) and the space for `TFTP_CONTEXT`.

Chapter 25

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.

25.1 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

25.2 Feature list

- Low memory footprint.
- Support PAP authentication protocol
- Support for PPP over Ethernet.

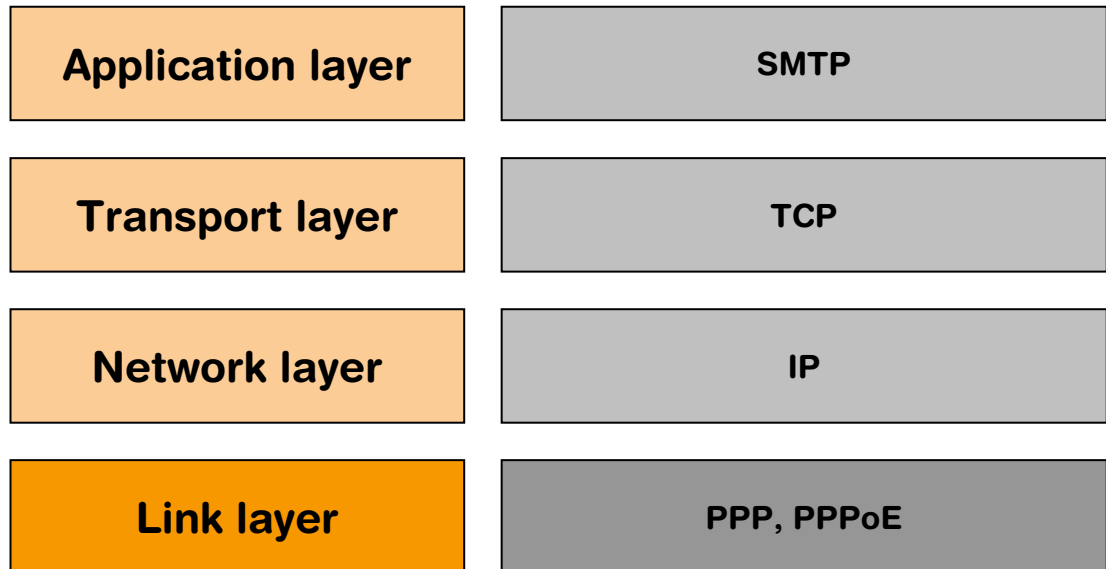
25.3 Requirements

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>".

25.4 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

25.5 API functions

Function	Description
PPPoE functions	
<code>IP_PPPOE_AddInterface()</code>	Adds a PPPoE interface.
<code>IP_PPPOE_ConfigRetries()</code>	Configures the number of times to resend a lost message before breaking the connection.
<code>IP_PPPOE_Reset()</code>	Resets the PPPoE connection.
<code>IP_PPPOE_SetAuthInfo()</code>	Sets the authentication information for the PPPoE connection.
<code>IP_PPPOE_SetUserCallback()</code>	Sets a callback function to inform the user about a status change of the connection.
PPP functions	
<code>IP_PPP_AddInterface()</code>	Adds a PPP driver.
<code>IP_PPP_OnRx()</code>	Receives one or more characters from the hardware.
<code>IP_PPP_OnRxChar()</code>	Receives a character from the hardware.
<code>IP_PPP_OnTxChar()</code>	Sends a character via PPP.
<code>IP_PPP_SetUserCallback()</code>	Sets a callback function to inform the user about a status change of the connection.
Modem functions	
<code>IP_MODEM_Connect()</code>	Connects using the modem line.
<code>IP_MODEM_Disconnect()</code>	Disconnects the modem line.
<code>IP_MODEM_GetResponse()</code>	Retrieves the last received responses from the modem.
<code>IP_MODEM_SendString()</code>	Sends a command to the modem.
<code>IP_MODEM_SendStringEx()</code>	Sends a command to the modem and checks for the correct response.
<code>IP_MODEM_SetAuthInfo()</code>	Sets authentication information required by your ISP.
<code>IP_MODEM_SetConnectTimeout()</code>	Sets the timeout how long to wait until the modem is fully connected.
<code>IP_MODEM_SetInitCallback()</code>	Sets a callback proving modem initialization.
<code>IP_MODEM_SetInitString()</code>	Sets a single command needed for modem initialization.
<code>IP_MODEM_SetSwitchToCmdDelay()</code>	Sets a delay when sending "+++ATH" is problematic.

Table 25.1: embOS/IP PPP/PPPoE/Modem API function overview

25.6 PPPoE functions

25.6.1 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 25.2: IP_PPPOE_AddInterface() parameter list

Return value

>= 0 Index of the interface.

25.6.2 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] Timeout in ms before a resend is triggered.

Table 25.3: IP_PPPOE_ConfigRetries() parameter list

25.6.3 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 25.4: IP_PPPOE_Reset() parameter list

25.6.4 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 25.5: IP_PPPOE_SetAuthInfo() parameter list

25.6.5 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 25.6: 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);
```


25.7 PPP functions

25.7.1 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 25.7: IP_PPP_AddInterface() parameter list

Return value

>= 0 Index of the interface.

25.7.2 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
<code>pContext</code>	[IN] Pointer to a Structure IP_PPP_CONTEXT .
<code>pData</code>	[IN] Pointer to a buffer which is storing the received data.
<code>NumBytes</code>	[IN] Number of received bytes.

Table 25.8: IP_PPP_OnRx() parameter list

25.7.3 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 25.9: IP_PPP_OnRxChar() parameter list

25.7.4 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

```
int IP_PPP_OnTxChar( unsigned Unit );
```

Parameter

Parameter	Description
Unit	Typically 0.

Table 25.10: IP_PPP_OnTxChar() parameter list

Return value

0: More data has been sent. Keep Tx interrupt enabled.
1: No more data to send. Disable Tx interrupt if necessary.

25.7.5 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 25.11: 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);
```

25.8 Modem functions

25.8.1 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
<code>sATCommand</code>	[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 25.12: IP_MODEM_Connect() parameter list

Return value

0: Connected
!= 0: Error

Example

```
IP_MODEM_Connect( "ATD*99***1#" );
```


25.8.2 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 25.13: IP_MODEM_Disconnect() parameter list

Example

```
IP_MODEM_Disconnect(0);
```

25.8.3 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.
pNumBytesInBuffer	[OUT] Number of bytes in receive buffer. May be NULL.

Table 25.14: 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);
```

25.8.4 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 25.15: IP_MODEM_SendString() parameter list

Example

```
IP_MODEM_SendString(0, "AT");
```

Additional information

This routine is meant for sending simple AT commands to the modem that do not need to be checked for their response.

It is not designed to be used with *IP_MODEM_GetResponse()* on page 658. If you intend to process the modem response please use *IP_MODEM_SendStringEx()* on page 660 instead.

25.8.5 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 multiple 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 25.16: IP_MODEM_SendStringEx() parameter list

Return value

0: O.K., correct 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` is 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 receive buffer.
//
if (IP_MEMCMP(IP_MODEM_GetResponse(0, NULL, 0, NULL), "+CPIN: READY", 12) != 0) {
    IP_Panic("Unrecognized response from modem.");
}
}
}

```

25.8.6 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 25.17: 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.

25.8.7 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
<code>IFaceId</code>	[IN] Zero-based interface index.
<code>ms</code>	[IN] Timeout in ms. Default: 15s.

Table 25.18: IP_MODEM_SetConnectTimeout() parameter list

Example

```
IP_MODEM_SetConnectTimeout(0, 30000);
```

25.8.8 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
<code>pfInit</code>	[IN] Void callback routine for initialization of the modem before connecting.

Table 25.19: 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#");
```


25.8.9 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
<code>sInit</code>	[IN] Command to be sent to the modem before connecting.

Table 25.20: IP_MODEM_SetInitString() parameter list

Example

```
IP_MODEM_SetInitString("ATE0V1");  
IP_MODEM_Connect("ATD*99***1#");
```

25.8.10 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
<code>IFaceId</code>	[IN] Zero-based interface index.
<code>ms</code>	[IN] Timeout in ms between sending "+++" and "ATH".

Table 25.21: 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.

25.9 PPP data structures

Function	Description
<code>IP_PPP_CONTEXT</code>	Structure which stores the information about the PPP connection.
<code>RESEND_INFO</code>	A structure which stores the resend condition for different stages of the PPP connection.
<code>IP_PPP_LINE_DRIVER</code>	Structure with pointers to application related functions.

Table 25.22: embOS/IP PPP data structure overview

25.9.1 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 address of the PPP interface.
IPCP.aDNSServer	An IP_ADDR to store the IP address 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 25.23: 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 25.23: Structure IP_PPP_CONTEXT member list

25.9.2 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;
#ifdef 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 25.24: Structure RESEND_INFO member list

25.9.3 Structure IP_PPP_LINE_DRIVER

Description

Structure with pointers to application related functions.

Prototype

```
typedef struct {
    void (*pfInit) (struct IP_PPP_CONTEXT * pPPContext);
    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 25.25: Structure IP_PPP_LINE_DRIVER member list

25.10 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.

25.10.1 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 7.0Kbyte

Table 25.26: PPPoE ROM usage ARM7

25.10.2 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 6.5Kbyte

Table 25.27: PPPoE ROM usage Cortex-M3

25.10.3 RAM usage

Addon	RAM
embOS/IP PPP used for PPPoE	approximately 100 bytes

Table 25.28: PPPoE RAM usage

25.11 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.

25.11.1 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	approximately 7.0 kBytes

Table 25.29: PPP ROM usage ARM7

25.11.2 RAM usage

Addon	RAM
embOS/IP PPP	approximately 0.5 kBytes

Table 25.30: PPP RAM usage

Chapter 26

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.

26.1 embOS/IP NetBIOS

The embOS/IP NetBIOS implementation is an optional extension which can be seamlessly integrated into your 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 Concepts 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

26.2 Feature list

- Low memory footprint.
- Seamless integration with the embOS/IP stack.
- Client based NetBIOS name resolution.

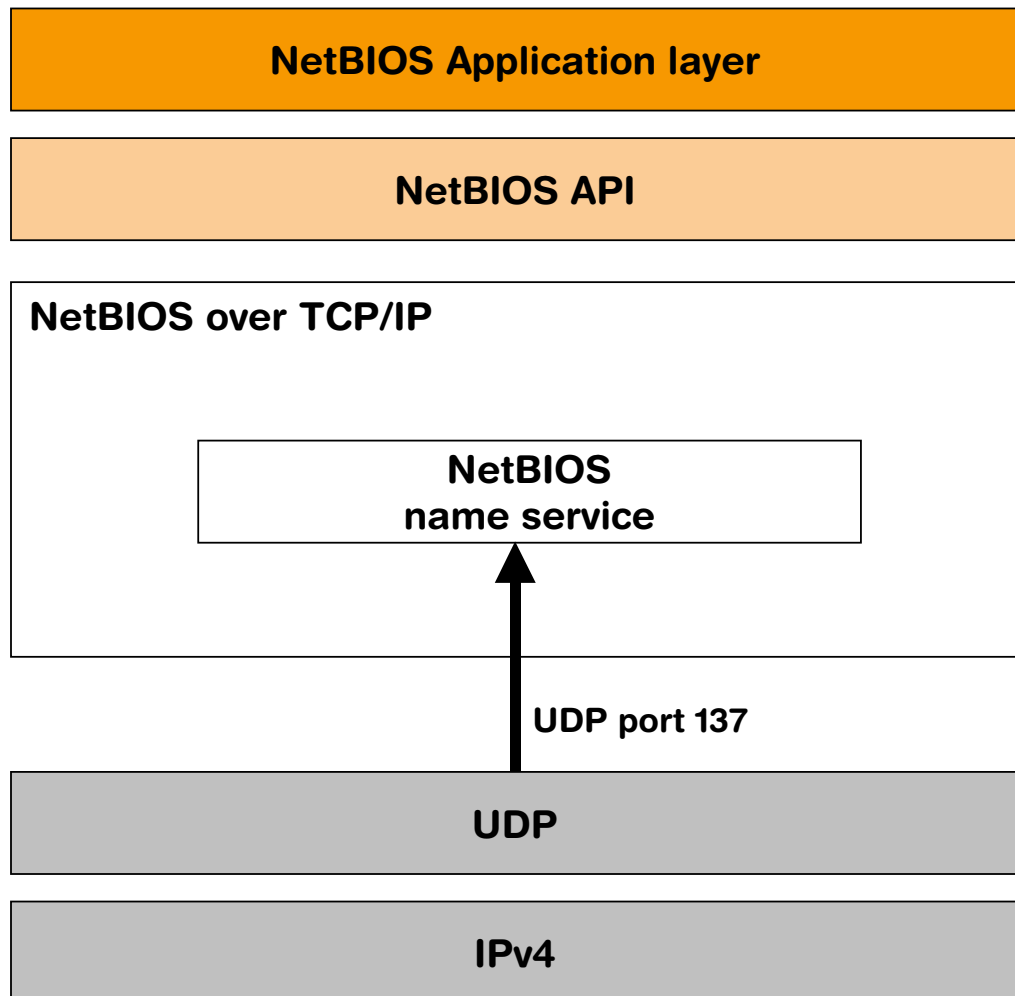
26.3 Requirements

TCP/IP stack

The embOS/IP NetBIOS implementation requires the embOS/IP TCP/IP stack.

26.4 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 area network via the session layer.



Using NetBIOS, an embOS/IP application can resolve a NetBIOS name to an IP address in the local area network.

26.5 API functions

Function	Description
NetBIOS	
<code>IP_NETBIOS_Init()</code>	Initializes the NetBIOS Name Service client.
<code>IP_NETBIOS_Start()</code>	Starts the NetBIOS client.
<code>IP_NETBIOS_Stop()</code>	Stops the NetBIOS client.

Table 26.1: embOS/IP NetBIOS API function overview

26.5.1 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 26.2: 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.

26.5.2 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 26.3: 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.

26.5.3 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 26.4: IP_NETBIOS_Stop() parameter list

26.5.4 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 26.5: Structure IP_NETBIOS_NAME member list

26.6 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.

26.6.1 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 26.6: NetBIOS ROM usage ARM7

26.6.2 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 26.7: NetBIOS ROM usage Cortex-M3

26.6.3 RAM usage

Addon	RAM
embOS/IP NetBIOS module	approximately 2.4Kbyte

Table 26.8: NetBIOS RAM usage

Chapter 27

SNTP client (Add-on)

The embOS/IP implementation of the Simple Network Time Protocol (SNTP) client is an optional extension to embOS/IP. It can be used to request a timestamp with the current time from a NTP server. All functions that are required to add SNTP client functionality to your application are described in this chapter.

27.1 embOS/IP SNTP client

The embOS/IP SNTP client implementation is an optional extension which can be seamlessly integrated into your application. It combines a maximum of performance with a small memory footprint. The SNTP client implementation allows an embedded system to use real timestamps from a remote NTP server without using a RTC or to initialize a RTC. The SNTP protocol is based on SNTP v4.

The SNTP client module implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 4330]	Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc4330.txt
[RFC 1305]	Network Time Protocol (Version 3) - Specification, Implementation and Analysis Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc1305.txt

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

Directory	Content
IP	Contains the SNTPc sources, IP_SNTPC.c.

Supplied directory structure of embOS/IP SNTPc package

27.2 Feature list

- Low memory footprint.
- Seamless integration with the embOS/IP stack.
- Time synchronization with a remote NTP server.

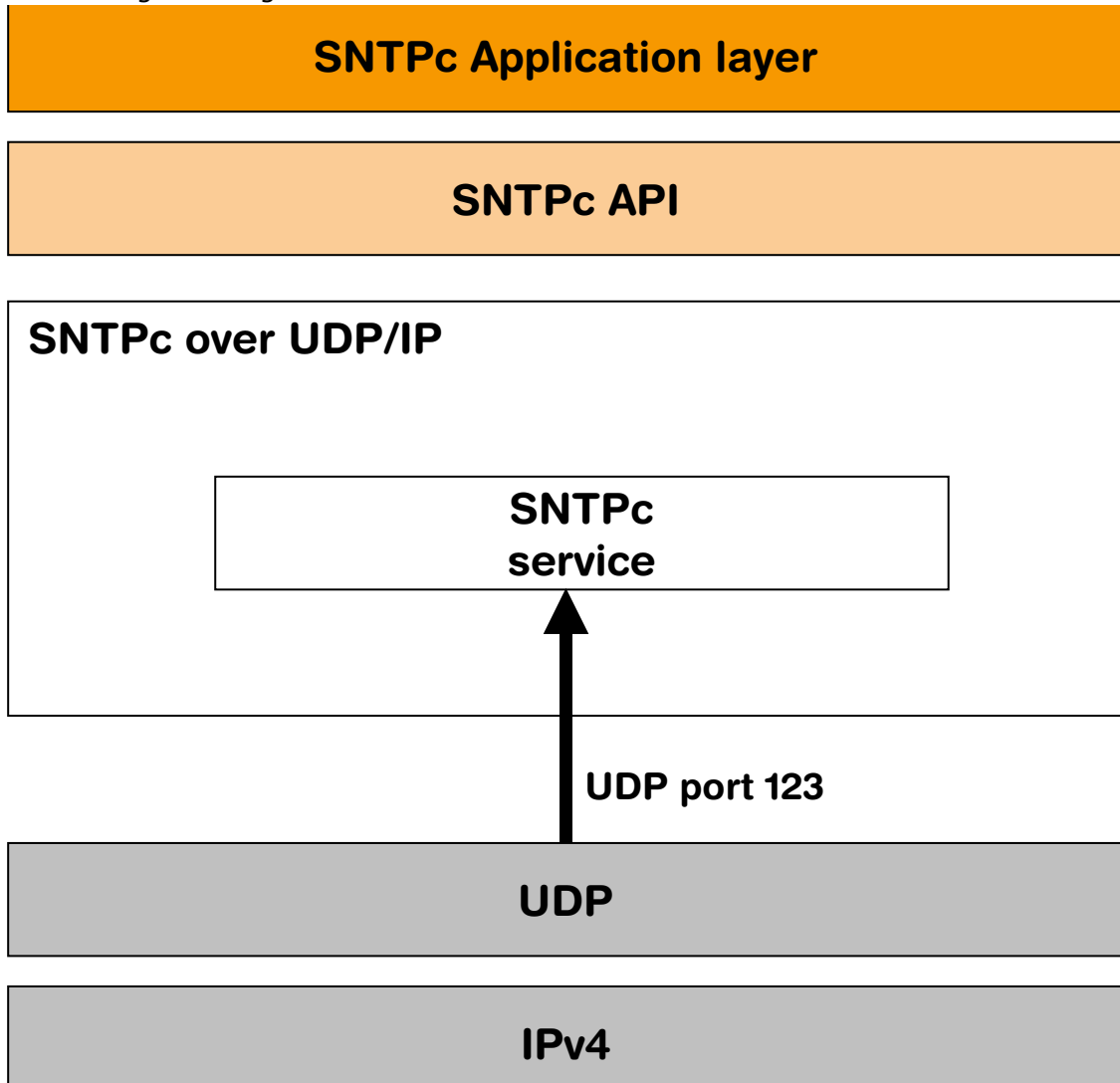
27.3 Requirements

TCP/IP stack

The embOS/IP SNTpC implementation requires the embOS/IP TCP/IP stack.

27.4 SNTP backgrounds

The SNTP protocol is an API on top of the TCP/IP protocol, it provides a way of synchronizing the target time with a local or remote NTP server over the network.



Using SNTP, an embOS/IP application can synchronize its time with a NTP server either in the local network or in a remote network to use a timestamp with the current date and time or to initialize its own RTC with a good start value.

27.4.1 The NTP timestamp

The NTP timestamp used is represented by a 64-bit value consisting of two 32-bit fields. The first 32-bit field contains the complete seconds passed since January 1st 1900. The second 32-bit field contains fractions of a second in 232 picoseconds.

More information about the NTP timestamp can be found in [RFC 1305](#).

27.4.2 The epoch problem (year 2036 problem)

The NTP timestamp reserves only 32-bit for full seconds passed which equals a little bit more than 136 years. As the NTP time is based on January 1st 1900 this means that the timestamp will overlap back to 0 some time in 2036. A timestamp older than a reference timestamp can be interpreted as valid time as well as long as it does not count up to the reference timestamp.

Based on this solution there are several possible ways of extending this period even more:

- The simplest solution to extend the timestamp to be used for around 136 years is for the target to remember the date it was built or has its firmware changed and can then use this timestamp as reference extending the NTP timestamp for further 136 years.
- Storing the current year in non volatile memory using it as reference in which epoch the target runs.
- Using other sources as reference for the epoch such as timestamps from other sources.

27.5 API functions

Function	Description
SNTP client	
<code>IP_SNTPC_ConfigTimeout()</code>	Configures request timeout.
<code>IP_SNTPC_GetTimeStampFromServer()</code>	Request timestamp from server.

Table 27.1: embOS/IP SNTPC API function overview

27.5.1 IP_SNTPC_ConfigTimeout()

Description

Configures the maximum time to wait for a response from a NTP server for a sent request.

Prototype

```
void IP_SNTPC_ConfigTimeout ( unsigned ms );
```

Parameter

Parameter	Description
<code>ms</code>	[IN] Timeout in ms to wait for a server response when requesting a timestamp.

Table 27.2: IP_SNTPC_ConfigTimeout() parameter list

27.5.2 IP_SNTPC_GetTimeStampFromServer()

Description

Request the current timestamp from a NTP server using the SNTP protocol.

Prototype

```
int IP_SNTPC_GetTimeStampFromServer( unsigned      IFaceId,
                                     const char    * sServer,
                                     IP_SNTP_TIMESTAMP * pTimestamp );
```

Parameter

Parameter	Description
IFaceId	[IN] Zero-based index of available network interfaces.
sServer	[IN] String containing either dotted decimal IP addr. (192.168.1.1) or DNS name (us.pool.ntp.org) of NTP server.
pTimestamp	[OUT] Pointer to an element of Structure IP_NTP_TIMESTAMP .

Table 27.3: IP_SNTPC_GetTimeStampFromServer() parameter list

Return value

Label	Numeric	Description
IP_SNTPC_STATE_NO_ANSWER	0	Request sent but no answer from server received within timeout.
IP_SNTPC_STATE_UPDATED	1	Timestamp updated from server response.
IP_SNTPC_STATE_KOD	2	Server sent Kiss-Of-Death and wants us to use another server.
Other	< 0	Error.

Table 27.4: IP_SNTPC_GetTimeStampFromServer() return value list

27.5.3 Structure IP_NTP_TIMESTAMP

Description

A structure which stores the timestamp from a NTP request.

Prototype

```
typedef struct IP_NTP_TIMESTAMP {
    U32 Seconds;
    U32 Fractions;
} IP_NTP_TIMESTAMP;
```

Member	Description
Seconds	Seconds passed since start of epoch, typically January 1st 1900.
Fractions	Fractions of a second in 232 picoseconds.

Table 27.5: Structure IP_NTP_TIMESTAMP member list

27.6 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.

27.6.1 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 SNTP client module	approximately 0.5Kbyte

Table 27.6: SNTP client ROM usage ARM7

27.6.2 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 SNTP client module	approximately 0.5Kbyte

Table 27.7: SNTP client ROM usage Cortex-M3

27.6.3 RAM usage

Addon	RAM
embOS/IP NetBIOS module	approximately 24 bytes

Table 27.8: SNTP client RAM usage

Chapter 28

SNMP agent (Add-on)

The embOS/IP Simple Network Management Protocol (SNMP) agent is an optional extension to embOS/IP. The SNMP agent can be used with embOS/IP or with a different UDP/IP stack. All functions that are required to add an SNMP agent to your application are described in this chapter.

Although SNMP can grow very complex very fast, the embOS/IP SNMP agent aims to be easily usable for anyone, even if you do not have too much in depth knowledge about SNMP at all. Therefore the API is kept to a minimum while still providing everything necessary for a full SNMP agent implementation.

28.1 embOS/IP SNMP agent

The embOS/IP SNMP agent is an optional extension which adds SNMP agent functionality to the stack. It combines a maximum of performance with a small memory footprint. The SNMP agent allows an embedded system to handle SNMP requests from an SNMP manager and sending TRAP and INFORM messages to managers. It comes with all features typically required by embedded systems: Maximum flexibility, easily portable and low RAM usage. RAM usage has been kept to a minimum by smart buffer handling.

The SNMP agent implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 2578]	Structure of Management Information Version 2 (SMIV2) Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc2578.txt
[RFC 3416]	Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP) Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc3416.txt
[RFC 4181]	Guidelines for Authors and Reviewers of MIB Documents Direct download: ftp://ftp.rfc-editor.org/in-notes/rfc4181.txt

RFCs used for embOS/IP SNMP agent package implementation

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

Directory	Content
.\Application\	Contains the example application to run the SNMP agent with embOS/IP.
.\Config\	Contains the SNMP agent configuration file. Refer to <i>Configuration</i> on page 719 for detailed information.
.\Inc\	Contains the required include files.
.\IP\	Contains the SNMP agent sources and header files, <code>IP_SNMP_AGENT*</code> .
.\Windows\IP\SNMP_Agent\	Contains the source, the project files and an executable to run the embOS/IP SNMP agent on a Microsoft Windows host. Refer to <i>Using the SNMP agent samples</i> on page 712 for detailed information.

Supplied directory structure of embOS/IP SNMP agent package

28.2 Feature list

- Low memory footprint.
- Only few SNMP knowledge required.
- Easy MIB tree setup.
- Supports SNMPv1 and SNMPv2c.
- Supports SNMPv1 & SNMPv2 TRAP messages.
- Supports SNMPv2 INFORM messages.
- MIB-II support (System and Interfaces branch) for embOS/IP out of the box.
- Easy to use API for all typical SNMP types (Unsigned32, Counter32, ...).
- Independent of the TCP/IP stack: any stack with sockets can be used.
- Can even be used without sockets e.g. with zero-copy API.
- Demo with custom sample MIB with sockets or zero-copy API included.
- Project for executable on PC for Microsoft Visual Studio included.

28.3 Requirements

TCP/IP stack

The embOS/IP SNMP agent requires an UDP/IP stack. It is optimized for embOS/IP, but any RFC-compliant UDP/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 as well as an implementation which uses the zero-copy API of embOS/IP.

IANA Private Enterprise Number (PEN)

For implementing SNMP in your own product you need to acquire a `Private Enterprise Number (PEN)` from the IANA. A PEN can be requested free of charge at the following location:

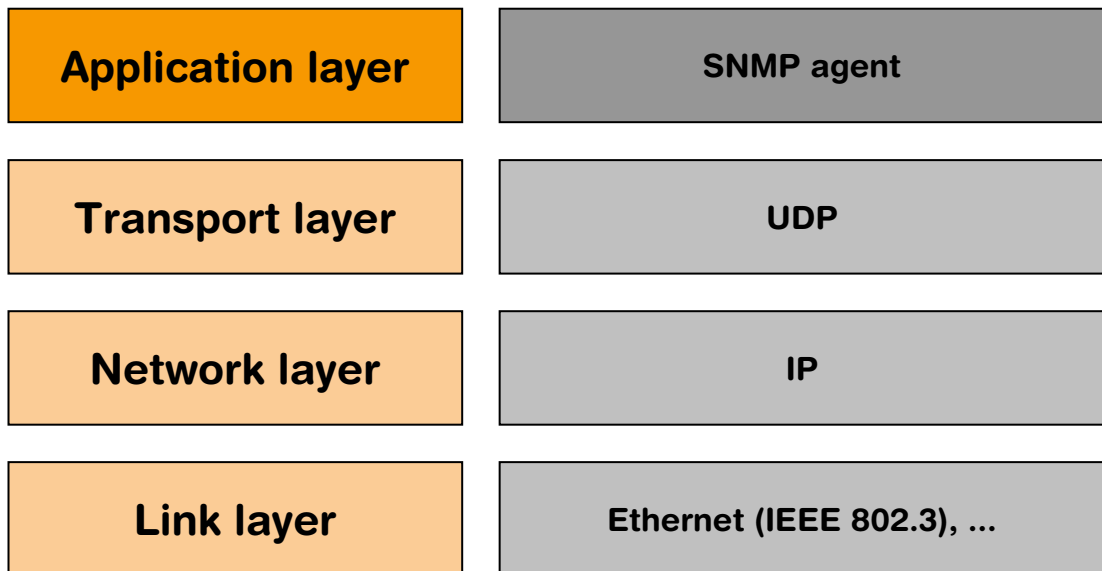
<http://pen.iana.org/pen/app>

The PEN used in the samples (dec. 46410) is the PEN registered for SEGGER Microcontroller GmbH & Co. KG. This needs to be changed to your own PEN in your product and the content of the MIB is subject to change in the future.

28.4 SNMP backgrounds

The Simple Network Management Protocol is a standard protocol to manage IP based devices in a network. Typical usage is to monitor counters and status in network switches and other network equipment but can also be used for configuration read/write operations. The development of SNMP is coordinated by the IETF (Internet Engineering Task Force) with the widely used SNMPv2c inspired from several other sources. The current protocol version supported by the embOS/IP SNMP agent is v1 and v2c. The latest protocol version is v3.

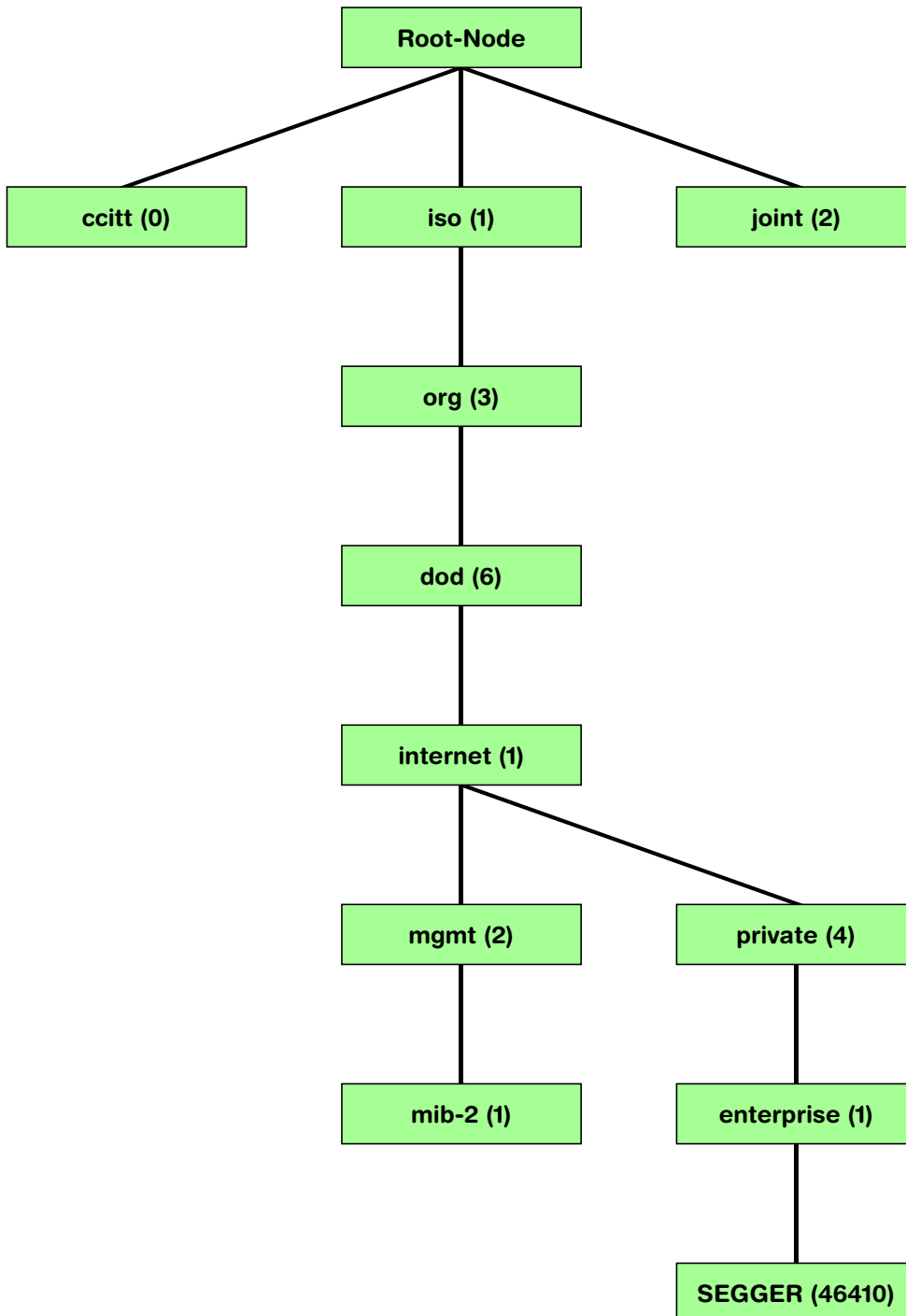
The SNMP agent is the part that is implemented directly inside a device to fulfill requests like status information and configuration of the device.



28.4.1 Data organization in SNMP

SNMP data is organized in Management Information Base (MIB) blocks. The blocks itself are typically called MIB itself to keep it short, instead of "MIB blocks". Each of these MIBs is organized in a tree like structure, able to have one parent and multiple childs. Every MIB has an unique Object Identifier (OID) on its level, making it possible to exactly walk the tree from top to bottom by following one MIB child OID after another.

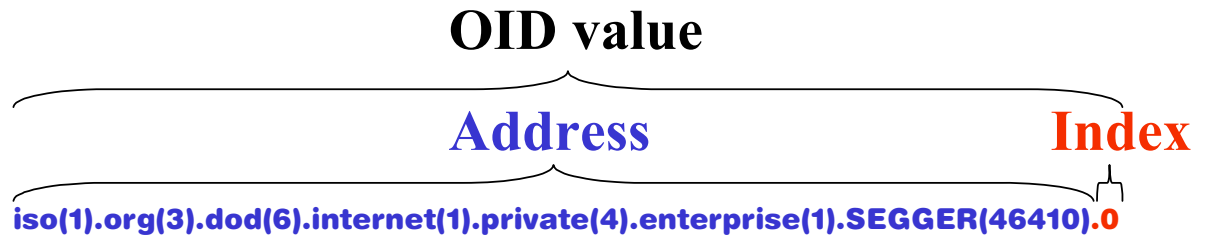
A typical MIB tree for access to your own enterprise specific data is shown below. In this example the MIBs from the root node down to the SEGGER enterprise MIB are shown.



As can be seen from the MIB tree above, the SEGGER MIB is located at 1.3.6.1.4.1.46410 . While everything above your own enterprise MIB is typically standardized, it is completely up to you what happens below your own MIB.

28.4.2 OID value, address and index

All locations and even items to access within the MIB tree are addressed by a so called "OID value". The `OID value` consists of multiple `OIDs` addressing one specific item of a specific MIB in the tree. For this purpose in general the `OID value` consists of two parts:



The address part describes which MIB in the tree needs to be addressed. The index specifies the element of the MIB to access. Typically index `.0` is used as scalar item which means that this is the main item of the addressed MIB. Typically this index is always present and serves the main feature of this MIB. For a hardware timer related MIB, index `.0` could contain the current value of the timer. However it completely depends on the MIB which purpose any index serves.

Although typically the last OID is used as index, the index is not limited to one OID, making it impossible to know the address length if not known from any other source like a MIB description. As it completely depends upon the implementor of the MIB how the MIB and addresses below this MIB are used, indexes can even be multi dimensional:



This can be used for example to access a table by column and row index.

28.4.3 SNMP data types

The SNMP standard defines various generic data types throughout its versions and several pseudo data types. These pseudo data types use the same type IDs as the data type they base on and from the pure data stream they can not be differentiated from their original type. They are used in MIB descriptions to give several items a clean meaning and boundaries that the original type does not support. However both sides need to be aware that this type and its characteristics are required for this specific item.

The following is a list of data types currently supported by the SNMP agent and includes the most common native data types and some of their more generic pseudo data types that are typically in use in the wild.

28.4.3.1 Native data types

These data types are types that in general are available since SNMPv1. Some of them have been renamed for SNMPv2 but in general still serve the same functionality and newer types are constructed from them.

Data Type	ID	Description
INTEGER	0x02	Signed 32-bit integer.
OCTET STRING	0x04	U8 data array.
OBJECT IDENTIFIER	0x06	OID value used as address to access or value like a pointer.
IpAddress	0x40	U32 IP address.
Counter	0x41	Unsigned 32-bit value, non decreasing. Allowed to be used with SNMPv2. Using the SNMPv2 Counter32 is advised when working with SNMPv2 only.
Gauge	0x42	Unsigned 32-bit value. Only SNMPv1.
Gauge32	0x42	Unsigned 32-bit value. Only SNMPv2 and above. Can be considered an alias of Gauge as its own type as only present for SNMPv2 and above.
TimeTicks	0x43	Unsigned 32-bit value, non decreasing.
Opaque	0x44	The Opaque type is typically only used in SMIV1 MIBs and can be used to extend the list of existing types with encapsulated and constructed types. The downside is that these types are not generic and can therefore not be understood by anyone else if they do not know the exact content encapsulated. The status with SNMPv2 for this type is deprecated but allowed.

Table 28.1: embOS/IP SNMP agent native data types overview

28.4.3.2 Constructed and new data types

These data types are either newer versions of an old SNMPv1 data type and have been renamed to be more precise in their name or they are technically the same as other native data types but respect other boundaries in their values. For all types the same rule applies: Both, manager and agent need to be aware of how to handle this data type and its characteristics.

Data Type	ID	Description
Integer32	0x02	Signed 32-bit integer. SNMPv2 version of the INTEGER data type.
BITS	0x04	Array of bit values sent in an U8 data array. Basically the same as the OCTET STRING data type.
Counter32	0x41	Unsigned 32-bit value, non decreasing. SNMPv2 version of Counter. Only supported by SNMPv2 and above.
Unsigned32	0x42	Unsigned 32-bit value. Only SNMPv2. Basically the same as the Gauge data type.
Opaque based data types		
Counter64	0x46	Unsigned 64-bit value, non decreasing. Implemented using the Opaque type.
Float	0x78	IEEE 754 single-precision float. Implemented using the Opaque type.
Double	0x79	IEEE 754 double-precision float. Implemented using the Opaque type.
Integer64	0x7A	Signed 64-bit integer. Implemented using the Opaque type.
Unsigned64	0x7B	Unsigned 64-bit value. Implemented using the Opaque type.

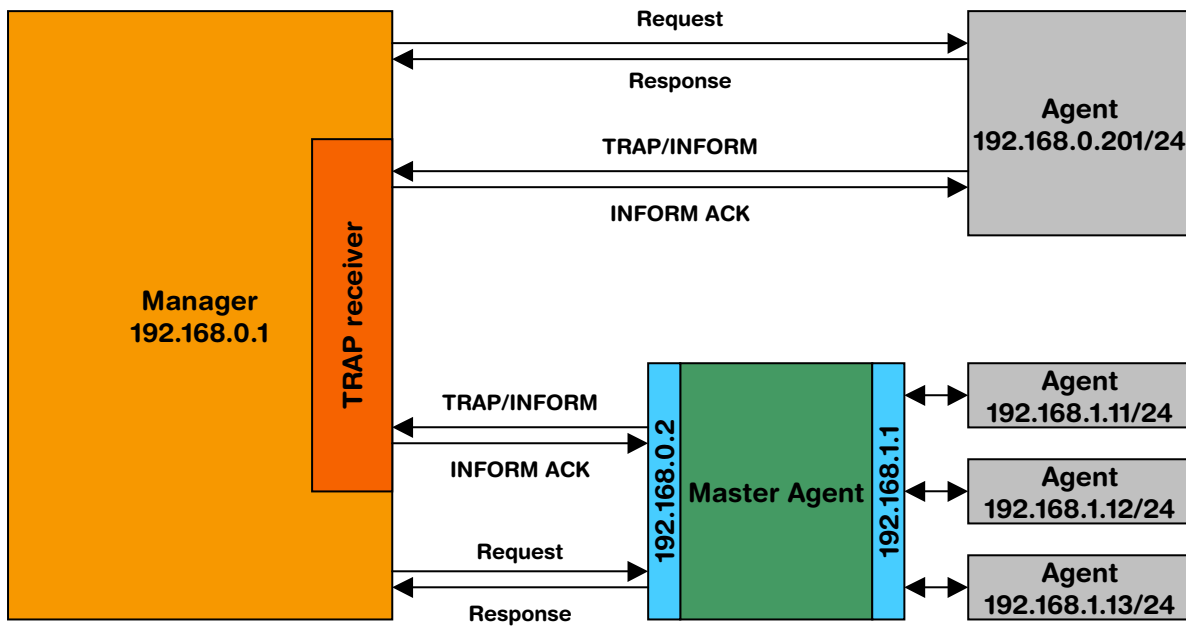
Table 28.2: embOS/IP SNMP agent constructed and new data types overview

28.4.4 Participants in an SNMP environment

In an SNMP environment there are typically at least an SNMP agent serving requests and a manager sending requests to the agent and waiting a response sent back. The only exception to this behavior is a TRAP (SNMPv1/SNMPv2c) message that is being sent unrequested from agent to manager to signal an event has happened. While for a TRAP message no answer is sent back from a manager to the agent, SNMPv2c introduces the INFORM (for SNMPv2c) message that awaits an acknowledge sent back from manager to agent.

While a single manager can operate with several agents alone, sometimes it is more efficient to use a master agent in between. The job of the master agent is to collect information from several agents and to provide a single communication partner for the manager to operate with. A manager can operate with multiple single agents and master agents at the same time.

The following picture shows the topology between a manager, one master agent and multiple agents:



28.4.5 Differences between SNMP versions

Today there are three SNMP versions that are used in products and can be considered as standards when talking about SNMP.

SNMPv1

The initial version of the protocol starting from 1988 supports the following message types:

Type	Description
<code>get-request</code>	Requests one or more values from specific OID values.
<code>getnext-request</code>	Requests the next available OID value after the start OID value sent with the getnext-request. Multiple requests can be sent in one getnext-request message.
<code>get-response</code>	Responses sent back upon any request message received.
<code>set-request</code>	Sets one or more values for specific OID values.
<code>TRAPv1</code>	Unrequested message sent from agent to one or more managers. Does not check for reception of the message.

Table 28.3: SNMPv1 message types

Although criticized for its poor security, only using a community string that is transmitted in cleartext SNMPv1 became the de facto standard for device management in a network.

SNMPv2c

SNMPv2 has been designed with better security in mind and supports the following new message types:

Type	Description
<code>getbulk-request</code>	Used for sending one or multiple getnext-requests to retrieve a large amount of data.
<code>TRAPv2</code>	Unrequested message sent from agent to one or more managers. Does not check for reception of the message. Same as a TRAPv1 message but does not use its own message format anymore but the standard SNMP PDU message format.
<code>INFORM</code>	Unrequested message sent from agent to one or more managers. Awaits an acknowledge sent back from the manager and is retransmitted if the acknowledge is not received.

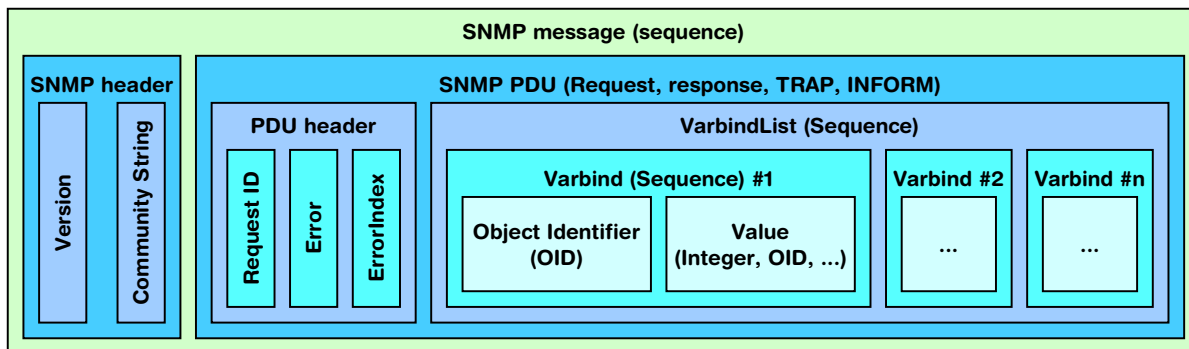
Table 28.4: SNMPv2 message types

The original SNMPv2 introduced a new security system which was seen by many as too complex and therefore not widely accepted. The de facto standard later became SNMPv2c which stands for SNMPv2 with community based security which works the same as in SNMPv1 by using a clear-type community string. SNMPv2c therefore combines the simple to use and implement community based security model with the improved performance and new message types introduced with SNMPv2.

28.4.6 SNMP communication basics

SNMP is a request and response protocol. This means that always a manager sends a request to an agent and the agent sends back a response to the manager. The only exception from this concept is for TRAP and INFORM messages where the agent sends a notification about an event that has happened at the agent to a defined list of one or more managers.

The encoding used in SNMP messages is a binary encoding, so the data can not be read clear-text. An SNMP message consists of several nested elements as can be seen in the following picture:



Typically each of these nested elements consists of a type field, a length field and the value. This applies to the "Object Identifier" and the "Value" in a Varbind in the same way.

28.4.7 SNMP agent return codes

The embOS/IP SNMP agent API distinguishes two types of return codes.

- Generic SNMP protocol error codes
- embOS/IP SNMP agent API error codes

Both types might be returned by the embOS/IP SNMP agent API depending on if an internal error in the API or an SNMP protocol error has occurred.

Generic SNMP protocol return codes

These return codes are generic for the SNMP protocol and are sent in an SNMP message. Only these return codes are expected to be used in a MIB callback registered with the embOS/IP SNMP agent API.

Return code	Value	Description
<code>IP_SNMP_OK</code>	0	Everything O.K.
<code>IP_SNMP_ERR_TOO_BIG</code>	1	Either the request received was too big to parse or the response to send does not fit into one SNMP message.
<code>IP_SNMP_ERR_NO_SUCH_NAME</code>	2	The OID value to access does not exist.
<code>IP_SNMP_ERR_BAD_VALUE</code>	3	Syntax or value error.
<code>IP_SNMP_ERR_GENERIC</code>	5	Generic error, not further specified.
<code>IP_SNMP_ERR_NO_ACCESS</code>	6	The OID value to access is not available. Might be due to not being visible for the supplied community string.
<code>IP_SNMP_ERR_WRONG_TYPE</code>	7	Typically sent back for a set-request with a non-matching type field for a value to set. For example if an INTEGER is expected but the set-request contains an Unsigned32 value to set.
<code>IP_SNMP_ERR_NO_CREATION</code>	11	The OID value to access in a set-request does not exist and the agent is unable to create it.
<code>IP_SNMP_ERR_AUTH</code>	16	No access to this OID value, e.g. wrong community string.

Table 28.5: SNMP generic protocol return codes

embOS/IP SNMP agent API return codes

These return codes are used by the embOS/IP SNMP agent API to return the result of internal processes such as parsing results and generating responses.

Return code	Value	Description
<code>IP_SNMP_AGENT_OK</code>	0	Everything O.K.
<code>IP_SNMP_AGENT_ERR_MISC</code>	-1	Generic error, not further specified.
<code>IP_SNMP_AGENT_ERR_UNSUPPORTED_VERSION</code>	-2	A message has been received with an unsupported protocol version.
<code>IP_SNMP_AGENT_ERR_AUTH</code>	-3	Wrong community string received in request. Requested action is not allowed.
<code>IP_SNMP_AGENT_ERR_MALFORMED_MESSAGE</code>	-4	Error while parsing a received message.

Table 28.6: embOS/IP SNMP agent return codes

28.5 Using the SNMP agent samples

Ready to use examples for Microsoft Windows and embOS/IP are supplied. If you use another UDP/IP stack, the samples have to be adapted.

The supplied sample application `OS_IP_SNMP_Agent.c` and `OS_IP_SNMP_Agent_ZeroCopy.c` basically do the same. They open the UDP ports 161 and 162 to listen for incoming SNMP messages. Port 161 is the default port for SNMP requests and is mandatory for SNMP communication. Port 162 is the default port for sending SNMP TRAP messages and receiving SNMP INFORM messages and the acknowledge sent back. Therefore in an SNMP agent if no INFORM messages shall be sent at all, listening on this port is optional.

The two samples supplied for embOS/IP do the same for the SNMP agent. However they differ in the way they are using the UDP/IP stack used for communication.

28.5.1 OS_IP_SNMP_Agent.c

This sample uses the standard BSD socket interface and can be easily ported to any other BSD socket compatible UDP/IP stack. It requires static or allocated buffers for most of the SNMP processing and typically causes several data copy operations in the IP stack itself due to the nature of the socket interface.

SNMP messages received need to be copied into a local buffer that can then be supplied to the SNMP agent API for message parsing. A second local buffer needs to be supplied to store the response to send back. The response then needs to be copied into the socket buffer by the IP stack for the UDP message to send.

Sending a TRAP or INFORM message again requires several local buffers to form and send/receive messages.

28.5.2 OS_IP_SNMP_Agent_ZeroCopy.c

This sample uses the embOS/IP UDP zero-copy API to handle SNMP messages in a very effective way without unnecessary copy operations in the receive and send paths.

In addition to saving almost all local buffers that are required for SNMP message processing, the SNMP agent can be handled easily without any task at all which saves task stack as well.

Received SNMP messages are directly passed to the SNMP agent together with their packet descriptor to the message parser routines without the need to copy their UDP payload into another buffer. The same applies for the response output. An allocated packet buffer can be passed completely to the parser routines and the response is stored directly in the packet buffer. This way the response can be passed to embOS/IP for sending without first having to copy the payload into another buffer.

As the packet buffers are pre-allocated in embOS/IP upon initialization and the memory is not freed internally, this also prevents memory fragmentation while providing a flexible way to allocate memory for receiving and sending SNMP messages.

28.5.3 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 the embOS/IP SNMP agent. If you do not have the Microsoft compiler, a precompiled executable of the SNMP agent is also supplied.

Building the sample program

Open the workspace `SNMP_Agent.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.

28.5.4 Features of the sample application

The sample as shipped is configured for operating/simulating a system with 8 LEDs.

A base MIB tree is added providing a sample implementation of a custom MIB node at the SEGGER enterprise OID value "1.3.6.1.4.1.46410". This evaluates to the following OID path in clear text: "iso(1).org(3).dod(6).internet(1).private(4).enterprise(1).ENCODED(46410)".

The ID "46410" is the *Private Enterprise Number (PEN)* registered for SEGGER. All OIDs above the value "127" have to be BER encoded. You will need to apply for your own PEN with the IANA to avoid collisions with other enterprises. You are only allowed to design a product with SNMP with your own PEN. Registering for your own PEN is free of charge. For more information please refer to *IANA Private Enterprise Number (PEN)* on page 702.

The sample provided is able to retrieve and set the status of 8 LEDs. A callback registered to the OID "iso(1).org(3).dod(6).internet(1).private(4).enterprise(1).ENCODED(46410)" handles all further OIDs beneath the OID of the MIB with the callback. The LED is indexed with a single dimensional index. The following indexes are supported:

- .0: Read only. Status of all LEDs in one single byte. Bit 0 set in this byte means that the first LED is on. Bit 1 set in this byte means that the second LED is on.
- .1 to .8: Read/Write. The status of the LEDs with index 0 to 7 can be retrieved/set using the corresponding index. For a set-request a value of type INTEGER is expected. A value of 0 clears the LED while any other value sets the LED.

After the initialization of the SNMP agent the sample application sends a coldBoot trap with a bogus `Varbind` to the hosts configured in the configuration area at the top of the sample application. This is typically done to notify managers of the availability of this agent.

The sample is configured to use two community strings for authentication:

- "public": Read only access.
- "Segger": Read/write access.

28.5.5 Testing the sample application

For an easy test of the SNMP protocol the de facto standard SNMP tool set `Net-SNMP` can be used. It can be downloaded from the following location:

<http://www.net-snmp.org>

The following command line examples show how the SNMP functionality of the sample can be tested.

Clear the first LED

```
snmpset -v2c -c Segger <AgentIP> 1.3.6.1.4.1.46410.1 i 0
...
iso.3.6.1.4.1.46410.1 = INTEGER: 0
```

Set the second LED

```
snmpset -v2c -c Segger <AgentIP> 1.3.6.1.4.1.46410.2 i 1
...
iso.3.6.1.4.1.46410.2 = INTEGER: 1
```

Retrieve the status of all 8 LEDs in one byte

```
snmpget -v2c -c public <AgentIP> 1.3.6.1.4.1.46410.0
...
iso.3.6.1.4.1.46410.0 = INTEGER: 2
```

First LED (bit 0) is cleared, second LED (bit 1) is set, all other LEDs are cleared.

Get the next available value in MIB tree.

Retrieve the first element available starting from "iso(1).org(3).dod(6)".

```
snmpgetnext -v2c -c public <AgentIP> 1.3.6
...
iso.3.6.1.4.1.46410.0 = INTEGER: 2
```

Returns the next available element in the MIB tree which is the the current LED status.

Get the LED in single byte and all single LED status in one request.

Retrieve the LED status and output up to 9 elements after the LED status index.

```
snmpbulkget -v2c -c public -Cn1 -Cr9 <AgentIP> 1.3.6.1.4.1.46410 1.3.6.1.4.1.46410.0
...
iso.3.6.1.4.1.46410.0 = INTEGER: 2
iso.3.6.1.4.1.46410.1 = INTEGER: 0
iso.3.6.1.4.1.46410.2 = INTEGER: 1
iso.3.6.1.4.1.46410.3 = INTEGER: 0
iso.3.6.1.4.1.46410.4 = INTEGER: 0
iso.3.6.1.4.1.46410.5 = INTEGER: 0
iso.3.6.1.4.1.46410.6 = INTEGER: 0
iso.3.6.1.4.1.46410.7 = INTEGER: 0
iso.3.6.1.4.1.46410.8 = INTEGER: 0
iso.3.6.1.4.1.46410.8 = No more variables left in this MIB View (It is past the
end of the MIB tree)
```

A getbulk-request is a combination of multiple getnext-requests. In this example "-Cn1" defines that the first OID value of all OIDs given as parameter shall only retrieve one value. "-Cn2" would mean that the two first OID values given shall return only one value each. For all following OID values "-Cr9" defines that up to 9 values shall be retrieved for each OID value. In this example only one OID value follows the single value OID value(s) and as only the index .0 to .8 is available in the sample and we start at .0 with a getnext-request, this retrieves the values for the indexes .1 to .8 with an exception that we reached the end of the MIB tree searching for the next element at index .8 .

Testing TRAP and INFORM messages

The sample sends a `coldBoot` TRAP or INFORM message depending on the configuration set in the sample to selected SNMP managers.

For testing TRAP and INFORM messages you will need either a fully functional SNMP manager or any other software that is able to open the trap UDP port (typically 162) and listens for TRAP or INFORM messages to be sent by an agent.

One simple software to test TRAP and INFORM messages is the free MIB browser available from ManageEngine that can be downloaded from the following location:

<https://www.manageengine.com/products/mibbrowser-free-tool/trap-receiver.html>

It does not only provide a GUI driven SNMP manager to test the agent but also comes with a TRAP browser to collect and visualize TRAP and INFORM messages sent by the embOS/IP SNMP agent. You will need to configure the IP address of the host running the TRAP browser in the SNMP agent sample.

28.6 The MIB callback

To each MIB of the embos/IP SNMP agent MIB tree, a callback for handling sub OIDs and indexes can be registered. For each OID value to access the SNMP agent will search for the first available OID of a parent MIB and execute the callback assigned to it. This callback will then be given information about the OID value to access. It then needs to parse the information available in the received message and form a response if no error shall be sent back.

As for an SNMP agent internally getbulk-requests are handled the same way as get-next-requests, the SNMP agent will use getnext-requests for the callback to fulfill, even if it originally was a getbulk-request that has been received.

Therefore the three types that shall be handled by a MIB callback are:

- set-request
- get-request
- getnext-request

A sample implementation can be found in the provided samples in the function `_SNMP_Sample_cb()`:

```

/*****
 *
 *      _SNMP_Sample_cb()
 *
 *  Function description
 *      Callback handler that can be assign to one or more MIBs.
 *
 *  Parameters
 *      pContext      : Context for the current message and response.
 *      pUserContext: User specific context passed to the process message API.
 *      pMIB         : Pointer to the MIB that is currently accessed.
 *      MIBLen       : Length of the MIB that is currently accessed.
 *      pIndex       : Pointer to the encoded index of the OID.
 *      IndexLen     : Length of the encoded index in bytes.
 *      RequestType  : IP_SNMP_PDU_SET_REQUEST or
 *                   IP_SNMP_PDU_GET_REQUEST or
 *                   IP_SNMP_PDU_TYPE_GET_NEXT_REQUEST .
 *      VarType      : Type of variable that waits to be parsed for input data.
 *                   Only valid if RequestType is IP_SNMP_PDU_SET_REQUEST .
 *
 *  Return value
 *      Everything O.K.      : IP_SNMP_OK
 *      In case of an error: IP_SNMP_ERR_*
 *
 *  Additional information
 *      - pIndex might point to more than one index OID value e.g. when
 *        using multi dimensional arrays. In any case the index should
 *        be decoded before it is used to make sure values above 127
 *        are correctly used.
 *      - Parameters of a set-request need to be stored back with their new value.
 *      - The memory that pMIB and pIndex point to might be reused by store
 *        functions. If the data stored at their location is important you
 *        have to save them locally on your own. It is advised to do all
 *        checks at the beginning of the callback so you do not rely on these
 *        parameters while or after you use store functions.
 */
static int _SNMP_Sample_cb(      IP_SNMP_AGENT_CONTEXT* pContext,
                                void*                  pUserContext,
                                const U8*               pMIB,
                                U32                     MIBLen,
                                const U8*               pIndex,
                                U32                     IndexLen,
                                U8                      RequestType,
                                U8                      VarType) {

    I32 OnOff;
    U32 Index;

```

```

U32 NumBytesDecoded;
U8  LEDMask;

(void)pUserContext; // Context passed through the whole SNMP API by
                    // the application.
(void)pMIB;         // OID part with the address of the MIB found.
(void)MIBLen;       // Length of the MIB OID.

LEDMask             = 0;
NumBytesDecoded = IndexLen;

if (IP_SNMP_AGENT_DecodeOIDValue(pIndex, &NumBytesDecoded, &Index, &pIndex) != 0) {
    return IP_SNMP_ERR_GENERIC;
}
if (RequestType == IP_SNMP_PDU_TYPE_GET_NEXT_REQUEST) { // Handle getnext-request.
    //
    // A getnext-request has to provide the next indexed item after the
    // one addressed. As for this sample we expect to have only a one
    // dimensional index this is simply incrementing the read index by one.
    // For a getnext-request the callback has to store the OID value of the new item
    // as well.
    // In general the callback has the following options how to react:
    // 1) The callback is able to store the next item after the given index:
    //    The callback has to store the the OID and the value of the next item.
    // 2) The callback is NOT able to store the next item after the given index:
    //    The callback has to store an NA exception or to report an
    //    IP_SNMP_ERR_NO_CREATION error.
    //
    Index++;
}
//
// Do some checks.
//
if ((Index > 8) || // This sample is designed for an index
    // of 0..8 where 0 is the status of all
    // LEDs and 1..8 is a LED index.
    (IndexLen > NumBytesDecoded)) { // We expect to have only one index, if
    //there are more index bytes to parse this
    // means an error (trying to access .x.y
    // where only .x is available).

    if (IP_SNMP_AGENT_StoreInstanceNA(pContext) != 0) {
        return IP_SNMP_ERR_TOO_BIG;
    }
    return IP_SNMP_OK;
    //
    // As alternate IP_SNMP_ERR_NO_CREATION can be returned but
    // will abort processing of the VarbindList.
    //
    // return IP_SNMP_ERR_NO_CREATION; // This resource does not exist.
}
if (Index != 0) {
    LEDMask = (1 << (Index - 1));
}
//
// Process get-, getnext- or set-request.
//
if (RequestType == IP_SNMP_PDU_TYPE_SET_REQUEST) { // Handle set-request.
    //
    // Check that the parameter type of the variable is what we expect.
    //
    if (VarType != IP_SNMP_TYPE_INTEGER) {
        return IP_SNMP_ERR_WRONG_TYPE;
    }
    if (IP_SNMP_AGENT_ParseInteger(pContext, &OnOff) != 0) {
        return IP_SNMP_ERR_GENERIC;
    }
    //

```

```

// Set LED state based on index:
//   Index 0: Invalid.
//   Other   : Set LED state for one specific LED by one byte.
//
if (Index == 0) {
    return IP_SNMP_ERR_NO_ACCESS;
} else {
    //
    // Set status of one LED.
    //
    if (OnOff != 0) { // On ?
        _LEDState |= LEDMask;
        BSP_SetLED(Index - 1);
    } else {
        _LEDState &= (LEDMask ^ 255);
        BSP_ClrLED(Index - 1);
    }
    if (IP_SNMP_AGENT_StoreInteger(pContext, OnOff) != 0) {
        return IP_SNMP_ERR_TOO_BIG;
    }
}
} else { // Handle get-request.
    if (RequestType == IP_SNMP_PDU_TYPE_GET_NEXT_REQUEST) {
        //
        // Store OID of "next" item returned if this is for a getnext-request.
        // The value will be stored by the following code in case of a getnext-
        // and a get-request.
        //
        if (IP_SNMP_AGENT_StoreCurrentMibOidAndIndex(pContext, pContext, 1, Index) !=
0) {
            return IP_SNMP_ERR_TOO_BIG;
        }
    }
    //
    // Get LED state based on index:
    //   Index 0: Get LED state for all 8 possible LEDs in one byte.
    //   Other   : Get LED state for one specific LED in one byte.
    //
    if (Index == 0) {
        if (IP_SNMP_AGENT_StoreInteger(pContext, _LEDState) != 0) {
            return IP_SNMP_ERR_TOO_BIG;
        }
    } else {
        //
        // Return status of one LED.
        //
        if ((_LEDState & LEDMask) != 0) {
            OnOff = 1;
        } else {
            OnOff = 0;
        }
        if (IP_SNMP_AGENT_StoreInteger(pContext, OnOff) != 0) {
            return IP_SNMP_ERR_TOO_BIG;
        }
    }
}
}
return IP_SNMP_OK;
}

```

28.7 Configuration

The embOS/IP SNMP agent 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.

28.7.1 Configuration macro types

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).

28.7.2 Compile time configuration switches

Type	Symbolic name	Default	Description
F	<code>IP_SNMP_AGENT_WARN</code>	--	Defines a function to output warnings. In debug configurations (<code>DEBUG == 1</code>) <code>IP_SNMP_AGENT_WARN</code> maps to <code>IP_Warnf_Application()</code> .
F	<code>IP_SNMP_AGENT_LOG</code>	--	Defines a function to output logging messages. In debug configurations (<code>DEBUG == 1</code>) <code>IP_SNMP_AGENT_LOG</code> maps to <code>IP_Logf_Application()</code> .
B	<code>IP_SNMP_AGENT_SUPPORT_PANIC_CHECK</code>	Debug: 1 Release: 0	Defines if upon a critical error the execution shall be halted.
B	<code>IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES</code>	1	Defines if 64-bit types like double and relatives like float are supported.
N	<code>IP_SNMP_AGENT_WORK_BUFFER</code>	64	Defines the size of the buffer used to do internal work like assembling OID values for further usage.

Table 28.7: embOS/IP SNMP agent compile time switches

28.8 API functions

Function	Description
<code>IP_SNMP_AGENT_AddCommunity()</code>	Adds a community string for access rights management.
<code>IP_SNMP_AGENT_AddMIB()</code>	Adds a MIB to the tree.
<code>IP_SNMP_AGENT_AddInformReponseHook()</code>	Registers a hook that is called upon a state change of an INFORM item waiting for an ACK.
<code>IP_SNMP_AGENT_CheckInformStatus()</code>	Retrieves the status of an INFORM message sent.
<code>IP_SNMP_AGENT_DeInit()</code>	Deinitializes the SNMP agent.
<code>IP_SNMP_AGENT_Exec()</code>	Executes management tasks.
<code>IP_SNMP_AGENT_GetMessageType()</code>	Retrieves the message type of the message in the given buffer.
<code>IP_SNMP_AGENT_Init()</code>	Initializes the SNMP agent.
<code>IP_SNMP_AGENT_PrepareTrapInform()</code>	Prepares a TRAP/INFORM message.
<code>IP_SNMP_AGENT_ProcessInformResponse()</code>	Processes a response that has been received for a previously sent INFORM message.
<code>IP_SNMP_AGENT_ProcessRequest()</code>	Processes a received message and fills the output buffer with the response for sending back.
<code>IP_SNMP_AGENT_SendTrapInform()</code>	Sends a TRAP/INFORM message.
<code>IP_SNMP_AGENT_SetCommunityPerm()</code>	Sets permissions for a community profile.
Standard MIB tree setup functions	
<code>IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2Interfaces()</code>	Adds the base MIBs iso(1).org(3).dod(6).internet(1).ietf(2).mib2(1).interfaces(2) to the tree.
<code>IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2System()</code>	Adds the base MIBs iso(1).org(3).dod(6).internet(1).ietf(2).mib2(1).system(1) to the tree.
<code>IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetPrivateEnterprise()</code>	Adds the base MIBs iso(1).org(3).dod(6).internet(1).private(4).enterprise(1) to the tree.
Message construct functions	
<code>IP_SNMP_AGENT_CloseVarbind()</code>	Closes a previously opened Varbind after an OID and value pair has been stored and inserts the length into the Varbind header.
<code>IP_SNMP_AGENT_OpenVarbind()</code>	Prepares a Varbind in the output buffer of the given context.
<code>IP_SNMP_AGENT_StoreBits()</code>	Stores bits data into an SNMP message.
<code>IP_SNMP_AGENT_StoreCounter()</code>	Stores a Counter into an SNMP message.
<code>IP_SNMP_AGENT_StoreCounter32()</code>	Stores a Counter32 into an SNMP message.
<code>IP_SNMP_AGENT_StoreCounter64()</code>	Stores a Counter64 into an SNMP message.
<code>IP_SNMP_AGENT_StoreCurrentMibOidAndIndex()</code>	Stores the currently processed MIB OID into an output buffer of another or the same context and adds the given indexes to the output buffer as well.

Table 28.8: embOS/IP SNMP agent interface function overview

Function	Description
<code>IP_SNMP_AGENT_StoreDouble()</code>	Stores a double-precision float into an SNMP message.
<code>IP_SNMP_AGENT_StoreFloat()</code>	Stores a single-precision float into an SNMP message.
<code>IP_SNMP_AGENT_StoreGauge()</code>	Stores a Gauge into an SNMP message.
<code>IP_SNMP_AGENT_StoreGauge32()</code>	Stores a Gauge32 into an SNMP message.
<code>IP_SNMP_AGENT_StoreInstanceNA()</code>	Stores a Varbind exception into an SNMP message if an instance (index) addressed is not available.
<code>IP_SNMP_AGENT_StoreInteger()</code>	Stores an INTEGER into an SNMP message.
<code>IP_SNMP_AGENT_StoreInteger32()</code>	Stores an Integer32 into an SNMP message.
<code>IP_SNMP_AGENT_StoreInteger64()</code>	Stores an Integer64 into an SNMP message.
<code>IP_SNMP_AGENT_StoreIpAddress()</code>	Stores an IpAddress into an SNMP message.
<code>IP_SNMP_AGENT_StoreOctetString()</code>	Stores an octet string into an SNMP message.
<code>IP_SNMP_AGENT_StoreOID()</code>	Stores an OID into an SNMP message.
<code>IP_SNMP_AGENT_StoreOpaque()</code>	Stores an Opaque into an SNMP message.
<code>IP_SNMP_AGENT_StoreTimeTicks()</code>	Stores a 32-bit TimeTick into an SNMP message.
<code>IP_SNMP_AGENT_StoreUnsigned32()</code>	Stores an Unsigned32 into an SNMP message.
<code>IP_SNMP_AGENT_StoreUnsigned64()</code>	Stores an Unsigned64 into an SNMP message.
Message parsing functions	
<code>IP_SNMP_AGENT_ParseBits()</code>	Parses a bits field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseCounter()</code>	Parses a Counter field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseCounter32()</code>	Parses a Counter32 field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseCounter64()</code>	Parses a Counter64 field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseDouble()</code>	Parses a double field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseFloat()</code>	Parses a float field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseGauge()</code>	Parses a Gauge field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseGauge32()</code>	Parses a Gauge32 field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseInteger()</code>	Parses an INTEGER field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseInteger32()</code>	Parses an Integer32 field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseInteger64()</code>	Parses an Integer64 field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseIpAddress()</code>	Parses an IpAddr field (IPv4) out of an SNMP message.

Table 28.8: embOS/IP SNMP agent interface function overview (Continued)

Function	Description
<code>IP_SNMP_AGENT_ParseOctetString()</code>	Parses an OCTET STRING out of an SNMP message.
<code>IP_SNMP_AGENT_ParseOID()</code>	Parses an OID out of an SNMP message.
<code>IP_SNMP_AGENT_ParseOpaque()</code>	Parses an Opaque field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseTimeTicks()</code>	Parses a 32-bit TimeTick field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseUnsigned32()</code>	Parses an Unsigned32 field out of an SNMP message.
<code>IP_SNMP_AGENT_ParseUnsigned64()</code>	Parses an Unsigned64 field out of an SNMP message.
Helper functions	
<code>IP_SNMP_AGENT_DecodeOIDValue()</code>	Parses and decodes an OID value of max $(2^{32}) - 1$ into an U32 to work with.
<code>IP_SNMP_AGENT_EncodeOIDValue()</code>	Encodes an OID value of max $(2^{32}) - 1$ (U32) into a buffer.

Table 28.8: embOS/IP SNMP agent interface function overview (Continued)

28.8.1 IP_SNMP_AGENT_AddCommunity()

Description

Adds a community string for access rights management.

Prototype

```
void IP_SNMP_AGENT_AddCommunity(      IP_SNMP_AGENT_COMMUNITY* pCommunity,
                                     const char*          sCommunity,
                                     U32                    Len );
```

Parameter

Parameter	Description
pCommunity	Pointer to IP_SNMP_AGENT_COMMUNITY memory block.
sCommunity	Community string to add.
Len	Length of the community string without termination.

Table 28.9: IP_SNMP_AGENT_AddCommunity() parameter list

28.8.2 IP_SNMP_AGENT_AddMIB()

Description

Adds a MIB to the tree.

Prototype

```
int IP_SNMP_AGENT_AddMIB(const U8*          pParentOID,
                          U32          Len,
                          IP_SNMP_AGENT_MIB* pMIB,
                          IP_SNMP_AGENT_pfMIB pf,
                          U32          Id );
```

Parameter

Parameter	Description
pParentOID	Pointer to parent OID in MIB tree.
Len	Length of parent OID.
pMIB	Pointer to new MIB to add.
pf	Callback handler for this MIB.
Id	Actual identifier of the new OID.

Table 28.10: IP_SNMP_AGENT_AddMIB() parameter list

Return value

O.K.: 0
Error: Other

28.8.3 IP_SNMP_AGENT_AddInformReponseHook()

Description

Registers a hook that is called upon a state change of an INFORM item waiting for an ACK.

Prototype

```
void IP_SNMP_AGENT_AddInformReponseHook(
    IP_SNMP_AGENT_HOOK_ON_INFORM_RESPONSE* pHook,
    IP_SNMP_AGENT_pfOnInformResponse      pf );
```

Parameter

Parameter	Description
pHook	Pointer to element of type IP_SNMP_AGENT_HOOK_ON_INFORM_RESPONSE.
pf	Function pointer to callback to hook in.

Table 28.11: IP_SNMP_AGENT_AddInformReponseHook() parameter list

28.8.4 IP_SNMP_AGENT_CancelInform()

Description

Cancels an INFORM message that might be still in process of resending. Resources for this message can then be freed.

Prototype

```
void IP_SNMP_AGENT_CancelInform (
    IP_SNMP_AGENT_TRAP_INFORM_CONTEXT* pTrapInformContext );
```

Parameter

Parameter	Description
pTrapInformContext	Pointer to context of message sent.

Table 28.12: IP_SNMP_AGENT_CancelInform() parameter list

Additional information

Canceling a message will not overwrite an already set status.

28.8.5 IP_SNMP_AGENT_CheckInformStatus()

Description

Retrieves the status of an INFORM message sent.

Prototype

```
int IP_SNMP_AGENT_CheckInformStatus(
    IP_SNMP_AGENT_TRAP_INFORM_CONTEXT* pContext );
```

Parameter

Parameter	Description
pContext	Pointer to INFORM context that has been used for sending.

Table 28.13: IP_SNMP_AGENT_CheckInformStatus() parameter list

Return value

Current status of the INFORM message:

```
IP_SNMP_AGENT_INFORM_STATUS_WAITING_FOR_ACK
IP_SNMP_AGENT_INFORM_STATUS_ACK_RECEIVED
IP_SNMP_AGENT_INFORM_STATUS_NACK_RECEIVED
IP_SNMP_AGENT_INFORM_STATUS_CANCELED
IP_SNMP_AGENT_INFORM_STATUS_TIMEOUT
```

28.8.6 IP_SNMP_AGENT_DeInit()

Description

Deinitializes the SNMP agent.

Prototype

```
void IP_SNMP_AGENT_DeInit(void);
```

Additional information

All tasks and external resources that use the SNMP agent API need to be stopped before deinitialization.

28.8.7 IP_SNMP_AGENT_Exec()

Description

Executes management tasks.

Prototype

```
I32 IP_SNMP_AGENT_Exec(void);
```

Return value

Time [ms] until the next timeout runs out.

Additional information

Typically checks if it is time to resend an INFORM message for which we have not yet received an ACK. The return value describes how long a task calling this function can sleep before it should execute this function again to handle management functionality.

28.8.8 IP_SNMP_AGENT_GetMessageType()

Description

Retrieves the message type of the message in the given buffer.

Prototype

```
int IP_SNMP_AGENT_GetMessageType (const U8* pIn,  
                                   U32 NumBytesIn,  
                                   U8* pType );
```

Parameter

Parameter	Description
pIn	Pointer to received SNMP message.
NumBytesIn	Length of received SNMP message.
pType	Pointer where to store the parsed PDU message type.

Table 28.14: IP_SNMP_AGENT_GetMessageType() parameter list

Return value

O.K. : 0

Error: Other

Additional information

By checking the message type beforehand it is possible to use the same port for normal get-requests and INFORM handling by using this routine to distinguish which API is required to process the received message.

28.8.9 IP_SNMP_AGENT_Init()

Description

Initializes the SNMP agent.

Prototype

```
void IP_SNMP_AGENT_Init( const IP_SNMP_AGENT_API* pAPI );
```

Parameter

Parameter	Description
pAPI	Pointer to SNMP agent API.

Table 28.15: IP_SNMP_AGENT_Init() parameter list

28.8.10 IP_SNMP_AGENT_PrepareTrapInform()

Description

Prepares a TRAP/INFORM message.

Prototype

```
void IP_SNMP_AGENT_PrepareTrapInform(
                                IP_SNMP_AGENT_CONTEXT* pContext,
                                void*                    pUserContext,
                                const U8*                pEnterpriseOID,
                                U32                      EnterpriseOIDLen,
                                const U8*                pTrapOID,
                                U32                      TrapOIDLen,
                                U8*                      pBuffer,
                                U32                      BufferSize,
                                U32                      AgentAddr );
```

Parameter

Parameter	Description
<code>pContext</code>	Pointer to an SNMP agent context.
<code>pUserContext</code>	User specific context passed to callbacks.
<code>pEnterpriseOID</code>	Pointer to enterprise OID sent in TRAP/INFORM.
<code>EnterpriseOIDLen</code>	Length of enterprise OID.
<code>pTrapOID</code>	Pointer to SMIV2 TRAP OID sent as source.
<code>TrapOIDLen</code>	Length of TRAP OID.
<code>pBuffer</code>	Pointer to buffer where to construct the Varbinds to send.
<code>BufferSize</code>	Size of the construct buffer.
<code>AgentAddr</code>	IP addr. of agent sending the traps. For IPv6 simply 0.

Table 28.16: IP_SNMP_AGENT_PrepareTrapInform() parameter list

Additional information

This routine prepares the context so that by using store functions a custom VarbindList can be build. The message can then be sent either once or multiple times using this context with the send routine.

For SNMPv1 and SNMPv2 TRAP/INFORM messages there are different information sent in them. What both share is that at least either the enterprise OID value or the TRAP OID value are included. The biggest difference is that SNMPv1 TRAPs did not have a location in the MIB tree and were a separate message type with its own message format that differs from all other SNMP messages. With SNMPv2 TRAPs now have a location inside the MIB tree.

For sending SNMPv1 and sending SNMPv2 messages there are different rules that apply but can be easily satisfied in your application by always providing both `pEnterpriseOID/EnterpriseOIDLen` and `pTrapOID/TrapOIDLen`. The information required for the specific SNMP version to send will be chosen automatically.

The SNMP agent API expects the TRAP OID value to be in SMIV2 form which means that the second to last OID is a zero. This is due to the fact that it is simpler to convert an SMIV2 OID value to SMIV1 than the other way round. For an SNMPv2 TRAP/INFORM the TRAP OID value is sent as it is as no conversion is necessary. For sending an SNMPv1 TRAP the TRAP OID value needs to be converted.

For a specific SNMPv1 TRAP this means that the last OID of the TRAP OID value is written into the `SpecificID` field of the message and all OIDs before the second to last zero OID are used as the `enterprise` OID value. However there is one exception to this procedure:

For a generic SNMPv1 TRAP a special rule applies. As TRAPs are now mapped into the MIB tree as well there is a difference to their previous form of not having a location at all. Their SMIV2 form does not have the second to last zero OID and they are not directly converted from SMIV2 to SMIV1 TRAP OID values as others but instead are mapped to their previous `GenericIDs`. In case of a generic SNMPv1 TRAP this means that the `EnterpriseID` is taken as provided by `pEnterpriseOID/EnterpriseOIDLen` and the SMIV2 TRAP OID value provided via `pTrapOID/TrapOIDLen` is replaced by the proper SMIV1 TRAP OID value. The following table shows the list of defines available from the SNMP agent header file to be used with `pTrapOID/TrapOIDLen` for sending SNMPv1 generic TRAPs:

Define	SMIV2 OID
IP_SNMP_GENERIC_TRAP_OID_COLD_START	1.3.6.1.6.3.1.1.5.1
IP_SNMP_GENERIC_TRAP_OID_WARM_START	1.3.6.1.6.3.1.1.5.2
IP_SNMP_GENERIC_TRAP_OID_LINK_DOWN	1.3.6.1.6.3.1.1.5.3
IP_SNMP_GENERIC_TRAP_OID_LINK_UP	1.3.6.1.6.3.1.1.5.4
IP_SNMP_GENERIC_TRAP_OID_AUTHENTICATION_FAILURE	1.3.6.1.6.3.1.1.5.5
IP_SNMP_GENERIC_TRAP_OID_EGP_NEIGHBOR_LOSS	1.3.6.1.6.3.1.1.5.6

Table 28.17: embOS/IP SNMP agent SMIV2 TRAP OID value defines overview

28.8.11 IP_SNMP_AGENT_ProcessInformResponse()

Description

Processes a response that has been received for a previously sent INFORM message.

Prototype

```
int IP_SNMP_AGENT_ProcessInformResponse( const U8* pIn,  
                                         U32 NumBytesIn );
```

Parameter

Parameter	Description
pIn	Pointer to received SNMP message.
NumBytesIn	Length of received SNMP message.

Table 28.18: IP_SNMP_AGENT_ProcessInformResponse() parameter list

Return value

O.K.: 0

Error: Other

28.8.12 IP_SNMP_AGENT_ProcessRequest()

Description

Processes a received message and fills the output buffer with the response for sending back.

Prototype

```
int IP_SNMP_AGENT_ProcessRequest( const U8*   pIn,
                                   U32      NumBytesIn,
                                   U8*      pOut,
                                   U32      NumBytesOut,
                                   void*    pUserContext );
```

Parameter

Parameter	Description
pIn	Pointer to received SNMP message.
NumBytesIn	Length of received SNMP message.
pOut	Pointer to output buffer to store the SNMP response.
NumBytesOut	Size of buffer for response.
pUserContext	User specific context passed to callbacks.

Table 28.19: IP_SNMP_AGENT_ProcessRequest() parameter list

Return value

Length of response to send: > 0
 No response to send : 0
 Error : < 0

28.8.13 IP_SNMP_AGENT_SendTrapInform()

Description

Sends a TRAP/INFORM message.

Prototype

```
int IP_SNMP_AGENT_SendTrapInform(
    void*                                pContext,
    IP_SNMP_AGENT_CONTEXT*               pVarbindContext,
    IP_SNMP_AGENT_TRAP_INFORM_CONTEXT* pTrapInformContext );
```

Parameter

Parameter	Description
pContext	Send context, typically a socket.
pVarbindContext	Pointer to an SNMP agent context holding Varbinds to send.
pTrapInformContext	Pointer to context of message to send.

Table 28.20: IP_SNMP_AGENT_SendTrapInform() parameter list

Return value

Message	Return value	Description
TRAP	== 0	O.K., pContext and pTrapInformContext can be freed.
TRAP	< 0	Error (buffer not big enough ?), pContext and pTrapInformContext can be freed.
INFORM	== 1	O.K., pContext and pTrapInformContext need to be preserved for further usage.
INFORM	< 0	Error (buffer not big enough ?), pContext and pTrapInformContext need to be preserved for further usage.

Table 28.21: IP_SNMP_AGENT_SendTrapInform() return value list

Additional information

An error while sending the first INFORM message might not mean that a resend can not succeed. This might happen if at the moment of sending the first message from this routine no send buffer is available. However in a resend a buffer might be available.

To be on the safe side if resources might be freed or not *IP_SNMP_AGENT_CancelInform()* on page 726 should be called for the message in question. After this it is safe to free the resources.

28.8.14 IP_SNMP_AGENT_SetCommunityPerm()

Description

Sets permissions for a community profile.

Prototype

```
void IP_SNMP_AGENT_SetCommunityPerm(
                                IP_SNMP_AGENT_COMMUNITY* pCommunity,
                                const IP_SNMP_AGENT_PERM*  pPerm );
```

Parameter

Parameter	Description
pCommunity	Pointer to IP_SNMP_AGENT_COMMUNITY memory block.
pPerm	Pointer to permissions table to use for this community.

Table 28.22: IP_SNMP_AGENT_SetCommunityPerm() parameter list

28.8.15 IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2Interfaces()

Description

Adds the base MIBs iso(1).org(3).dod(6).internet(1).ietf(2).mib2(1).interfaces(2) to the tree.

Prototype

```
int IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2Interfaces (
    const IP_SNMP_AGENT_MIB2_INTERFACES_API* pAPI );
```

Parameter

Parameter	Description
pAPI	Pointer to information and callback structure of type IP_SNMP_AGENT_MIB2_INTERFACES_API.

Table 28.23: IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2Interfaces() parameter list

Return value

O.K.: 0
Error: Other

28.8.16 IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2System()

Description

Adds the base MIBs iso(1).org(3).dod(6).internet(1).ietf(2).mib2(1).system(1) to the tree.

Prototype

```
int IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2System (
    const IP_SNMP_AGENT_MIB2_SYSTEM_API* pAPI );
```

Parameter

Parameter	Description
pAPI	Pointer to information and callback structure of type IP_SNMP_AGENT_MIB2_SYSTEM_API.

Table 28.24: IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2System() parameter list

Return value

O.K.: 0

Error: Other

28.8.17 IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetPrivateEnterprise()

Description

Adds the base MIBs iso(1).org(3).dod(6).internet(1).private(4).enterprise(1) to the tree.

Prototype

```
int IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetPrivateEnterprise( void );
```

Return value

O.K.: 0

Error: Other

28.8.18 IP_SNMP_AGENT_CloseVarbind()

Description

Closes a previously opened Varbind after an OID and value pair has been stored and inserts the length into the Varbind header.

Prototype

```
int IP_SNMP_AGENT_CloseVarbind( IP_SNMP_AGENT_CONTEXT* pContext );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.

Table 28.25: IP_SNMP_AGENT_CloseVarbind() parameter list

Return value

O.K.: 0

Error: Other

28.8.19 IP_SNMP_AGENT_OpenVarbind()

Description

Prepares a Varbind in the output buffer of the given context.

Prototype

```
int IP_SNMP_AGENT_OpenVarbind( IP_SNMP_AGENT_CONTEXT* pContext );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.

Table 28.26: IP_SNMP_AGENT_OpenVarbind() parameter list

Return value

O.K.: 0

Error (buffer not big enough?): Other

28.8.20 IP_SNMP_AGENT_StoreBits()

Description

Stores bits data into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreBits(          IP_SNMP_AGENT_CONTEXT* pContext,
                                   const U8*                pData,
                                   U32                      NumBytes );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pData	Pointer to bits to store.
NumBytes	Length of bits data.

Table 28.27: IP_SNMP_AGENT_StoreBits() parameter list

Return value

O.K.: 0

Error: Other

Additional information

This is an alias to *IP_SNMP_AGENT_StoreOctetString()* on page 757. The same type ID is used by the SNMP standard.

28.8.21 IP_SNMP_AGENT_StoreCounter()

Description

Stores a Counter into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreCounter( IP_SNMP_AGENT_CONTEXT* pContext,
                                U32                      v );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
v	Counter to store.

Table 28.28: IP_SNMP_AGENT_StoreCounter() parameter list

Return value

O.K.: 0
Error: Other

Additional information

This is an alias to *IP_SNMP_AGENT_StoreCounter32()* on page 745. The same type ID is used by the SNMP standard.

28.8.22 IP_SNMP_AGENT_StoreCounter32()

Description

Stores a Counter32 into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreCounter32( IP_SNMP_AGENT_CONTEXT* pContext,
                                  U32                      v );
```

Parameter

Parameter	Description
<code>pContext</code>	Pointer to an SNMP agent context.
<code>v</code>	Counter32 to store.

Table 28.29: IP_SNMP_AGENT_StoreCounter32() parameter list

Return value

O.K.: 0

Error: Other

28.8.23 IP_SNMP_AGENT_StoreCounter64()

Description

Stores a Counter64 into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreCounter64( IP_SNMP_AGENT_CONTEXT* pContext,
                                  U64                      v );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
v	Counter64 to store.

Table 28.30: IP_SNMP_AGENT_StoreCounter64() parameter list

Return value

O.K.: 0
Error: Other

Additional information

Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES == 1.

28.8.24 IP_SNMP_AGENT_StoreCurrentMibOidAndIndex()

Description

Stores the currently processed MIB OID into an output buffer of another or the same context and adds the given indexes to the output buffer as well.

Prototype

```
int IP_SNMP_AGENT_StoreCurrentMibOidAndIndex(
    IP_SNMP_AGENT_CONTEXT* pDstContext,
    IP_SNMP_AGENT_CONTEXT* pSrcContext,
    U32                      NumIndexes,
    ... );
```

Parameter

Parameter	Description
pDstContext	Pointer to an SNMP agent context to store the OID value.
pSrcContext	Pointer to an SNMP agent context from where to generate the OID value.
NumIndexes	Number of variable arguments passed to this function.
...	Variable arguments list.

Table 28.31: IP_SNMP_AGENT_StoreCurrentMibOidAndIndex() parameter list

Return value

O.K.: 0

Error: Other

28.8.25 IP_SNMP_AGENT_StoreDouble()

Description

Stores a double-precision float into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreDouble( IP_SNMP_AGENT_CONTEXT* pContext,  
                               double                v );
```

Parameter

Parameter	Description
<code>pContext</code>	Pointer to an SNMP agent context.
<code>v</code>	Double to store.

Table 28.32: IP_SNMP_AGENT_StoreDouble() parameter list

Return value

O.K.: 0

Error: Other

Additional information

A value of type double is expected to be presented in IEEE 754 form. This is true for all compilers following the C99 standard and typically even for almost all other compilers following older C-standards.

An easy way to check this is to test that a variable of type float with value 1.0 is stored as 0x3FF00000 in memory.

Can only be used when `IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES == 1`.

28.8.26 IP_SNMP_AGENT_StoreFloat()

Description

Stores a single-precision float into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreFloat( IP_SNMP_AGENT_CONTEXT* pContext,
                             float v );
```

Parameter

Parameter	Description
<code>pContext</code>	Pointer to an SNMP agent context.
<code>v</code>	Float to store.

Table 28.33: IP_SNMP_AGENT_StoreFloat() parameter list

Return value

O.K.: 0

Error: Other

Additional information

A value of type float is expected to be presented in IEEE 754 form. This is true for all compilers following the C99 standard and typically even for almost all other compilers following older C-standards.

An easy way to check this is to test that a variable of type float with value 1.0 is stored as 0x3F800000 in memory.

28.8.27 IP_SNMP_AGENT_StoreGauge()

Description

Stores a Gauge into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreGauge( IP_SNMP_AGENT_CONTEXT* pContext,
                               U32                    v) ;
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
v	Gauge to store.

Table 28.34: IP_SNMP_AGENT_StoreGauge() parameter list

Return value

O.K.: 0
Error: Other

Additional information

This is an alias to *IP_SNMP_AGENT_StoreUnsigned32()* on page 761. The same type ID is used by the SNMP standard.

28.8.28 IP_SNMP_AGENT_StoreGauge32()

Description

Stores a Gauge32 into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreGauge32( IP_SNMP_AGENT_CONTEXT* pContext,
                                U32                    v);
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
v	Gauge32 to store.

Table 28.35: IP_SNMP_AGENT_StoreGauge32() parameter list

Return value

O.K.: 0

Error: Other

Additional information

This is an alias to *IP_SNMP_AGENT_StoreUnsigned32()* on page 761. The same type ID is used by the SNMP standard.

28.8.29 IP_SNMP_AGENT_StoreInstanceNA()

Description

Stores a Varbind exception into an SNMP message if an instance (index) addressed is not available.

Prototype

```
int IP_SNMP_AGENT_StoreInstanceNA( IP_SNMP_AGENT_CONTEXT* pContext );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.

Table 28.36: IP_SNMP_AGENT_StoreInstanceNA() parameter list

Return value

O.K.: 0

Error: Other

28.8.30 IP_SNMP_AGENT_StoreInteger()

Description

Stores an INTEGER into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreInteger( IP_SNMP_AGENT_CONTEXT* pContext,
                               I32                      v );
```

Parameter

Parameter	Description
<code>pContext</code>	Pointer to an SNMP agent context.
<code>v</code>	INTEGER to store.

Table 28.37: IP_SNMP_AGENT_StoreInteger() parameter list

Return value

O.K.: 0

Error: Other

28.8.31 IP_SNMP_AGENT_StoreInteger32()

Description

Stores an Integer32 into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreInteger32( IP_SNMP_AGENT_CONTEXT* pContext,
                                   I32                      v );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
v	Integer32 to store.

Table 28.38: IP_SNMP_AGENT_StoreInteger32() parameter list

Return value

O.K.: 0
Error: Other

Additional information

This is an alias to *IP_SNMP_AGENT_StoreInteger()* on page 753. The same type ID is used by the SNMP standard.

28.8.32 IP_SNMP_AGENT_StoreInteger64()

Description

Stores an Integer64 into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreInteger64( IP_SNMP_AGENT_CONTEXT* pContext,
                                  I64                      v );
```

Parameter

Parameter	Description
<code>pContext</code>	Pointer to an SNMP agent context.
<code>v</code>	Integer64 to store.

Table 28.39: IP_SNMP_AGENT_StoreInteger64() parameter list

Return value

O.K.: 0

Error: Other

Additional information

Can only be used when `IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES == 1`.

28.8.33 IP_SNMP_AGENT_StoreIpAddress()

Description

Stores an IpAddress into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreIpAddress( IP_SNMP_AGENT_CONTEXT* pContext,  
                                U32                      IpAddress );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
IpAddress	IPv4 addr. as U32 in host order to store.

Table 28.40: IP_SNMP_AGENT_StoreIpAddress() parameter list

Return value

O.K.: 0
Error: Other

28.8.34 IP_SNMP_AGENT_StoreOctetString()

Description

Stores an octet string into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreOctetString(      IP_SNMP_AGENT_CONTEXT* pContext,
                                         const U8* pData,
                                         U32 NumBytes );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pData	Pointer to octet string to store.
NumBytes	Length of octet string.

Table 28.41: IP_SNMP_AGENT_StoreOctetString() parameter list

Return value

O.K.: 0

Error: Other

28.8.35 IP_SNMP_AGENT_StoreOID()

Description

Stores an OID into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreOID(          IP_SNMP_AGENT_CONTEXT* pContext,
                                   const U8*                pOIDBytes,
                                   U32                        OIDLen,
                                   U32                        MIBLen,
                                   U8                         IsValue );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pOIDBytes	Pointer to OID value.
OIDLen	Length of OID value.
MIBLen	Length of OID value that is part of the MIB.
IsValue	0: This is the OID value for which the result is sent. 1: This is a value field that contains an OID value.

Table 28.42: IP_SNMP_AGENT_StoreOID() parameter list

Return value

O.K.: 0
Error: Other

28.8.36 IP_SNMP_AGENT_StoreOpaque()

Description

Stores an Opaque into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreOpaque(          IP_SNMP_AGENT_CONTEXT* pContext,
                                     const U8*                pData,
                                     U32                      NumBytes );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pData	Pointer to Opaque data to store.
NumBytes	Size of Opaque data.

Table 28.43: IP_SNMP_AGENT_StoreOpaque() parameter list

Return value

O.K.: 0

Error: Other

Additional information

As an Opaque is a complex type and the content can be anything this function provides only the Opaque type field and the length field. All other content like the type inside the Opaque and the value itself has to be provided by the application.

28.8.37 IP_SNMP_AGENT_StoreTimeTicks()

Description

Stores a 32-bit TimeTick into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreTimeTicks( IP_SNMP_AGENT_CONTEXT* pContext,
                                  U32                      v );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
v	TimeTicks to store.

Table 28.44: IP_SNMP_AGENT_StoreTimeTicks() parameter list

Return value

O.K.: 0
Error: Other

28.8.38 IP_SNMP_AGENT_StoreUnsigned32()

Description

Stores an Unsigned32 into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreUnsigned32( IP_SNMP_AGENT_CONTEXT* pContext,
                                   U32                      v );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
v	Unsigned32 to store.

Table 28.45: IP_SNMP_AGENT_StoreUnsigned32() parameter list

Return value

O.K.: 0
Error: Other

28.8.39 IP_SNMP_AGENT_StoreUnsigned64()

Description

Stores an Unsigned64 into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreUnsigned64( IP_SNMP_AGENT_CONTEXT* pContext,
                                   U64                      v );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
v	Unsigned64 to store.

Table 28.46: IP_SNMP_AGENT_StoreUnsigned64() parameter list

Return value

O.K.: 0
Error: Other

Additional information

Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES == 1.

28.8.40 IP_SNMP_AGENT_ParseBits()

Description

Parses a bits field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseBits( IP_SNMP_AGENT_CONTEXT* pContext,
                             U8**                  ppData,
                             U32*                  pLen);
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
ppData	Pointer where to store the pointer to the data in the message.
pLen	Pointer where to store the data len.

Table 28.47: IP_SNMP_AGENT_ParseBits() parameter list

Return value

O.K.: 0

Error: Other

Additional information

This function expects that the type field has not been eaten out of the buffer.

This is an alias to *IP_SNMP_AGENT_ParseOctetString()* on page 775. The same type ID is used by the SNMP standard.

28.8.41 IP_SNMP_AGENT_ParseCounter()

Description

Parses a Counter field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseCounter( IP_SNMP_AGENT_CONTEXT* pContext,
                                U32*                    pCounter32 );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pCounter32	Pointer where to store the Counter.

Table 28.48: IP_SNMP_AGENT_ParseCounter() parameter list

Return value

O.K.: 0
Error: Other

Additional information

By design a Counter is a 32-bit unsigned value which does not mean that always 4 bytes are used in a message.

This function expects that the type field has not been eaten out of the buffer.

This is an alias to *IP_SNMP_AGENT_ParseCounter32()* on page 765. The same type ID is used by the SNMP standard.

28.8.42 IP_SNMP_AGENT_ParseCounter32()

Description

Parses a Counter32 field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseCounter32( IP_SNMP_AGENT_CONTEXT* pContext,
                                  U32* pCounter32 );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pCounter32	Pointer where to store the parsed Counter32.

Table 28.49: IP_SNMP_AGENT_ParseCounter32() parameter list

Return value

O.K.: 0

Error: Other

Additional information

By design a Counter32 is a 32-bit unsigned value which does not mean that always 4 bytes are used in a message.

This function expects that the type field has not been eaten out of the buffer.

28.8.43 IP_SNMP_AGENT_ParseCounter64()

Description

Parses a Counter64 field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseCounter64( IP_SNMP_AGENT_CONTEXT* pContext,
                                  U64* pCounter64 );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pCounter64	Pointer where to store the parsed Counter64.

Table 28.50: IP_SNMP_AGENT_ParseCounter64() parameter list

Return value

O.K.: 0
Error: Other

Additional information

By design a Counter64 is a 64-bit unsigned value which does not mean that always 8 bytes are used in a message.
This function expects that the type field has not been eaten out of the buffer.
Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES == 1.

28.8.44 IP_SNMP_AGENT_ParseDouble()

Description

Parses a double field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseDouble( IP_SNMP_AGENT_CONTEXT* pContext,
                               double*                pDouble );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pDouble	Pointer where to store the parsed double.

Table 28.51: IP_SNMP_AGENT_ParseDouble() parameter list

Return value

O.K.: 0

Error: Other

Additional information

A value of type double is expected to be presented in IEEE 754 form. This is true for all compilers following the C99 standard and typically even for almost all other compilers following older C-standards.

An easy way to check this is to test that a variable of type double with value 1.0 is stored as 0x3FF0000000000000 in memory.

This function expects that the type field has not been eaten out of the buffer.

Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES == 1.

28.8.45 IP_SNMP_AGENT_ParseFloat()

Description

Parses a float field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseFloat( IP_SNMP_AGENT_CONTEXT* pContext,  
                             float* pFloat);
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pFloat	Pointer where to store the parsed float.

Table 28.52: IP_SNMP_AGENT_ParseFloat() parameter list

Return value

O.K.: 0

Error: Other

Additional information

A value of type float is expected to be presented in IEEE 754 form. This is true for all compilers following the C99 standard and typically even for almost all other compilers following older C-standards.

An easy way to check this is to test that a variable of type float with value 1.0 is stored as 0x3F800000 in memory.

This function expects that the type field has not been eaten out of the buffer.

28.8.46 IP_SNMP_AGENT_ParseGauge()

Description

Parses a Gauge field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseGauge( IP_SNMP_AGENT_CONTEXT* pContext,
                             U32* pUnsigned32 );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pUnsigned32	Pointer where to store the parsed Gauge.

Table 28.53: IP_SNMP_AGENT_ParseGauge() parameter list

Return value

O.K.: 0

Error: Other

Additional information

By design a Gauge is a 32-bit unsigned value which does not mean that always 4 bytes are used in a message.

This function expects that the type field has not been eaten out of the buffer.

This is an alias to *IP_SNMP_AGENT_ParseUnsigned32()* on page 779. The same type ID is used by the SNMP standard.

28.8.47 IP_SNMP_AGENT_ParseGauge32()

Description

Parses a Gauge32 field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseGauge32( IP_SNMP_AGENT_CONTEXT* pContext,  
                                U32* pUnsigned32 );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pUnsigned32	Pointer where to store the parsed Gauge32.

Table 28.54: IP_SNMP_AGENT_ParseGauge32() parameter list

Return value

O.K.: 0

Error: Other

Additional information

By design a Gauge32 is a 32-bit unsigned value which does not mean that always 4 bytes are used in a message.

This function expects that the type field has not been eaten out of the buffer.

This is an alias to *IP_SNMP_AGENT_ParseUnsigned32()* on page 779. The same type ID is used by the SNMP standard.

28.8.48 IP_SNMP_AGENT_ParseInteger()

Description

Parses an INTEGER field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseInteger( IP_SNMP_AGENT_CONTEXT* pContext,
                               I32* pInteger );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pInteger	Pointer where to store the parsed INTEGER.

Table 28.55: IP_SNMP_AGENT_ParseInteger() parameter list

Return value

O.K.: 0

Error: Other

Additional information

By design an INTEGER is a 32-bit signed value which does not mean that always 4 bytes are used in a message.

This function expects that the type field has not been eaten out of the buffer.

28.8.49 IP_SNMP_AGENT_ParseInteger32()

Description

Parses an Integer32 field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseInteger32( IP_SNMP_AGENT_CONTEXT* pContext,
                                  I32*                    pInteger );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pInteger	Pointer where to store the parsed Integer32.

Table 28.56: IP_SNMP_AGENT_ParseInteger32() parameter list

Return value

O.K.: 0
Error: Other

Additional information

By design an Integer32 is a 32-bit signed value which does not mean that always 4 bytes are used in a message.

This function expects that the type field has not been eaten out of the buffer.

This is an alias to *IP_SNMP_AGENT_ParseInteger()* on page 771. The same type ID is used by the SNMP standard.

28.8.50 IP_SNMP_AGENT_ParseInteger64()

Description

Parses an Integer64 field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseInteger64( IP_SNMP_AGENT_CONTEXT* pContext,
                                  I64*                    pInteger64 );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pInteger64	Pointer where to store the parsed Integer64.

Table 28.57: IP_SNMP_AGENT_ParseInteger64() parameter list

Return value

O.K.: 0

Error: Other

Additional information

By design an Integer64 is a 64-bit signed value which does not mean that always 8 bytes are used in a message.

This function expects that the type field has not been eaten out of the buffer.

Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES == 1.

28.8.51 IP_SNMP_AGENT_ParseIpAddress()

Description

Parses an IpAddr field (IPv4) out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseIpAddress( IP_SNMP_AGENT_CONTEXT* pContext,
                                  U32*                    pIpAddress );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pIpAddress	Pointer where to store the parsed IpAddr. The IP addr. is stored as U32 in host order.

Table 28.58: IP_SNMP_AGENT_ParseIpAddress() parameter list

Return value

O.K.: 0
Error: Other

Additional information

This function expects that the type field has not been eaten out of the buffer.

28.8.52 IP_SNMP_AGENT_ParseOctetString()

Description

Parses an OCTET STRING out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseOctetString( IP_SNMP_AGENT_CONTEXT* pContext,
                                   U8**                  ppData,
                                   U32*                   pLen);
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
ppData	Pointer where to store the pointer to the data in the message.
pLen	Pointer where to store the data len.

Table 28.59: IP_SNMP_AGENT_ParseOctetString() parameter list

Return value

O.K.: 0

Error: Other

Additional information

This function expects that the type field has not been eaten out of the buffer.

28.8.53 IP_SNMP_AGENT_ParseOID()

Description

Parses an OID out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseOID( IP_SNMP_AGENT_CONTEXT* pContext,
                           U8**                  ppData,
                           U32*                  pLen );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
ppData	Pointer where to store the pointer to the data in the message.
pLen	Pointer where to store the data len.

Table 28.60: IP_SNMP_AGENT_ParseOID() parameter list

Return value

O.K.: 0
Error: Other

Additional information

This function expects that the type field has not been eaten out of the buffer.

28.8.54 IP_SNMP_AGENT_ParseOpaque()

Description

Parses an Opaque field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseOpaque( IP_SNMP_AGENT_CONTEXT* pContext,
                               U8**                  ppData,
                               U32*                  pLen );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
ppData	Pointer where to store the pointer to the Opaque data in the message.
pLen	Pointer where to store the Opaque data len.

Table 28.61: IP_SNMP_AGENT_ParseOpaque() parameter list

Return value

O.K.: 0

Error: Other

Additional information

This function expects that the type field has not been eaten out of the buffer.

28.8.55 IP_SNMP_AGENT_ParseTimeTicks()

Description

Parses a 32-bit TimeTick field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseTimeTicks( IP_SNMP_AGENT_CONTEXT* pContext,  
                                  U32* pTimeTicks );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pTimeTicks	Pointer where to store the parsed TimeTicks.

Table 28.62: IP_SNMP_AGENT_ParseTimeTicks() parameter list

Return value

O.K.: 0
Error: Other

Additional information

This function expects that the type field has not been eaten out of the buffer.

28.8.56 IP_SNMP_AGENT_ParseUnsigned32()

Description

Parses an Unsigned32 field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseUnsigned32( IP_SNMP_AGENT_CONTEXT* pContext,
                                   U32*                    pUnsigned32 );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pUnsigned32	Pointer where to store the parsed Unsigned32.

Table 28.63: IP_SNMP_AGENT_ParseUnsigned32() parameter list

Return value

O.K.: 0

Error: Other

Additional information

By design an Unsigned32 is a 32-bit unsigned value which does not mean that always 4 bytes are used in a message.

This function expects that the type field has not been eaten out of the buffer.

28.8.57 IP_SNMP_AGENT_ParseUnsigned64()

Description

Parses an Unsigned64 field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseInteger64( IP_SNMP_AGENT_CONTEXT* pContext,
                                  U64*                    pUnsigned64 );
```

Parameter

Parameter	Description
pContext	Pointer to an SNMP agent context.
pUnsigned64	Pointer where to store the parsed Unsigned64.

Table 28.64: IP_SNMP_AGENT_ParseUnsigned64() parameter list

Return value

O.K.: 0
Error: Other

Additional information

By design an Unsigned64 is a 64-bit unsigned value which does not mean that always 8 bytes are used in a message.
This function expects that the type field has not been eatenout of the buffer.
Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES == 1.

28.8.58 IP_SNMP_AGENT_DecodeOIDValue()

Description

Parses and decodes an OID value of max $(2^{32}) - 1$ into an U32 to work with.

Prototype

```
int IP_SNMP_AGENT_DecodeOIDValue( const U8*  pOID,
                                   U32*  pLen,
                                   U32*  pValue,
                                   const U8** ppNext );
```

Parameter

Parameter	Description
pOID	Pointer to next OID value to decode.
pLen	In: Pointer to length of the OID at pOID. Out: Pointer where to store the number of bytes decoded.
pValue	Pointer where to store the decoded OID value.
ppNext	Pointer to the pointer to the next OID value after the one processed. Can be NULL.

Table 28.65: IP_SNMP_AGENT_DecodeOIDValue() parameter list

Return value

O.K.: 0

Error: Other

28.8.59 IP_SNMP_AGENT_EncodeOIDValue()

Description

Encodes an OID value of max $(2^{32}) - 1$ (U32) into a buffer.

Prototype

```
int IP_SNMP_AGENT_EncodeOIDValue( U32  Value,
                                   U8*  pBuffer,
                                   U32  BufferSize,
                                   U8** ppNext,
                                   U8*  pNumEncodedBytes );
```

Parameter

Parameter	Description
Value	OID value to encode and store at pBuffer.
pBuffer	Pointer where to store the encoded OID value.
BufferSize	Size of destination buffer.
ppNext	Pointer to the pointer to the next OID value after the one processed.
pNumEncodedBytes	Pointer where to store the number of encoded and stored bytes.

Table 28.66: IP_SNMP_AGENT_EncodeOIDValue() parameter list

Return value

O.K.: 0

Error: Other

Additional information

BufferSize needs to be big enough for the encoded OID value. If unsure use a buffer of 5 bytes as this is the maximum size of an encoded OID value or use at least enough bytes as required for the value in memory + 1 byte as a rough calculation.

Examples

127 will be encoded in one byte. + 1 byte just to be on the safe side.

128 will be encoded in two bytes. + 1 byte is required.

28.9 SNMP agent data structures

28.9.1 Structure IP_SNMP_AGENT_API

Description

Used to provide an interface to external functions required for proper function of the SNMP agent.

Prototype

```
typedef struct {
    void (*pfInit)                (void);
    void (*pfDeInit)             (void);
    void (*pfLock)               (void);
    void (*pfUnlock)             (void);
    void* (*pfAllocSendBuffer)   (void* pUserContext, U8** ppBuffer,
                                  U32 NumBytes, U8 IPAddrLen);
    void (*pfFreeSendBuffer)     (void* pUserContext, void* p,
                                  char SendCalled, int r);
    int (*pfSendTrapInform)      (void* pContext, void* pUserContext,
                                  void* hBuffer, const U8* pData,
                                  U32 NumBytes, U8* pIPAddr,
                                  U16 Port, U8 IPAddrLen);
    U32 (*pfGetTime)             (void);
    U32 (*pfSysTicks2SnmpTime)   (U32 SysTicks);
    U32 (*pfSnmpTime2SysTicks)   (U32 SnmpTime);
} IP_SNMP_AGENT_API;
```

Member	Description
pfInit	Callback for initialization required for any other callback such as pfLock/pfUnlock. Called from <i>IP_SNMP_AGENT_Init()</i> on page 731.
pfDeInit	Callback for deinitialization called from <i>IP_SNMP_AGENT_DeInit()</i> on page 728.
pfLock	Callback for API locking in a multitasking environment.
pfUnlock	Callback for API unlocking in a multitasking environment.
pfAllocSendBuffer	Callback for allocating a buffer to store a message that can be sent at a later time via pfSendTrapInform .
pfFreeSendBuffer	Callback for freeing a buffer previously allocated with pfAllocSendBuffer .
pfSendTrapInform	Callback for sending a previously allocated buffer that has been filled with a message to send.
pfGetTime	Callback to retrieve the current system time in milliseconds.
pfSysTicks2SnmpTime	Callback to convert the system time (typically 1ms) into SNMP time of 1/100 seconds since an epoch.
pfSnmpTime2SysTicks	Callback to convert an SNMP timestamp of 1/100 seconds since an epoch into the system time (typically 1ms).

Table 28.67: Structure IP_SNMP_AGENT_API member list

28.9.2 Structure IP_SNMP_AGENT_PERM

Description

Used to grant permissions to a community that has been added to the SNMP agent.

Prototype

```
typedef struct {
    const U8* pOID;
    U16 Len;
    U8 Perm;
} IP_SNMP_AGENT_PERM;
```

Member	Description
pOID	Pointer to OID value to grant access permissions.
Len	Length of OID located at pOID .
Perm	Permissions to grant at this OID value and its child OIDs.

Table 28.68: Structure IP_SNMP_AGENT_PERM member list

Additional information

Permissions for various OID values can be set by using an array of IP_SNMP_AGENT_PERM entries that can then be used with *IP_SNMP_AGENT_SetCommunityPerm()* on page 737. Even if there is only one permission rule to set, there has to be an additional entry that always needs to be present. This last entry will specify the default permissions to grant for every OID that can not inherit permissions from a parent rule.

The default entry works the same way as any other entry but it needs to be the last entry and [pOID/Len](#) are set to the values [NULL/0](#).

The permissions that can be set for [Perm](#) are an ORRed value of the following masks:

Define	Description
IP_SNMP_AGENT_PERM_READ_MASK	Grants read access to this community for the OID value specified in the entry and its child OIDs.
IP_SNMP_AGENT_PERM_WRITE_MASK	Grants write access to this community for the OID value specified in the entry and its child OIDs.

Table 28.69: embOS/IP SNMP agent IP_SNMP_AGENT_PERM permission defines overview

28.9.3 Structure IP_SNMP_AGENT_MIB2_SYSTEM_API

Description

System description represented at MIB-II at oid value 1.3.6.1.2.1.1 .

For more details please refer to

<http://www.alvestrand.no/objectid/1.3.6.1.2.1.1.html> .

A sample implementation can be found in the shipped SNMP agent samples.

Prototype

```
typedef struct {
    const char* sSysDescr;
    const U8*   pSysObjectID;
        U32   SysObjectIDLen;
    U32 (*pfGetSysUpTime)      (void);
    int (*pfGetSetSysContact) (char* pBuffer, U32* pNumBytes, char IsWrite);
    int (*pfGetSetSysName)    (char* pBuffer, U32* pNumBytes, char IsWrite);
    int (*pfGetSetSysLocation)(char* pBuffer, U32* pNumBytes, char IsWrite);
    U8   SysServices;
} IP_SNMP_AGENT_MIB2_SYSTEM_API;
```

Member	Description
sSysDescr	String including full name and version of the target and other information. Up to 255 characters + termination.
pSysObjectID	The vendor's authoritative identification of the network management subsystem contained in the entity.
SysObjectIDLen	Length of the oid value at pSysObjectID.
pfGetSysUpTime	Time in in hundredths of a second since the network management portion of the system was last re-initialized.
pfGetSetSysContact	String including information regarding the contact person for this managed node and how to contact this person. Up to 255 characters + termination.
pfGetSetSysName	String including an administratively-assigned name for this managed node e.g. FQDN. Up to 255 characters + termination.
pfGetSetSysLocation	String including the physical location of this node. Up to 255 characters + termination.
SysServices	Value representing the services offered.

Table 28.70: Structure IP_SNMP_AGENT_MIB2_SYSTEM_API member list

28.9.4 Structure IP_SNMP_AGENT_MIB2_INTERFACES_API

Description

Interfaces description represented at MIB-II at oid value 1.3.6.1.2.1.1 .

For more details please refer to

<http://www.alvestrand.no/objectid/1.3.6.1.2.1.1.html> .

A sample implementation for embOS/IP is shipped with the SNMP agent in the file `IP_SNMP_AGENT_MIB2_INTERFACES_embOSIP.c` . The members of the structure are based on the SNMP structure of MIB-II interface counters.

To enable statistic counters in embOS/IP to provide the SNMP MIB-II interfaces counters with valid values please enable `IP_SUPPORT_STATS_IFACE` .

28.10 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the SNMP agent 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 IP_SNMP_AGENT_WORK_BUFFER 64
```

28.10.1 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 SNMP agent	approximately 6.5Kbyte

Table 28.71: SNMP agent ROM usage ARM7

28.10.2 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 SNMP agent	approximately 6.0Kbyte

Table 28.72: SNMP agent ROM usage Cortex-M3

28.10.3 RAM usage:

The following resource usage shows typical RAM requirements for an SNMP agent implementation. Most of the RAM is consumed for building the MIB tree.

Addon	ROM
embOS/IP SNMP agent	approximately 300 bytes

Table 28.73: SNMP agent RAM usage

Chapter 29

Profiling with SystemView

This chapter describes how to configure and enable profiling of embOS/IP using SystemView.

29.1 Overview

embOS/IP is instrumented to generate profiling information of API functions and driver-level functions.

These profiling information expose the run-time behavior of embOS/IP in an application, recording which API functions have been called, how long the execution took, and revealing which driver-level functions have been called by API functions or events like interrupts.

The profiling information is recorded using SystemView.

SystemView is a real-time recording and visualization tool for profiling data. It exposes the true run-time behavior of a system, going far deeper than the insight provided by debuggers. This is particularly effective when developing and working with complex systems comprising an OS with multiple threads and interrupts, and one or more middleware components.

SystemView can ensure a system performs as designed, can track down inefficiencies, and show unintended interactions and resource conflicts.

The recording of profiling information with SystemView is minimally intrusive to the system and can be done on virtually any system. With SEGGER's Real Time Technology (RTT) and a J-Link, SystemView can record data in real-time and analyze the data live, while the system is running.

The embOS/IP profiling instrumentation can be easily configured and set up.

29.2 Additional files

The SystemView module needs to be added to the application to enable profiling. If not already part of the project, download the sources from <https://www.segger.com/systemview.html> and add them to the project.

Also make sure that `IP_SYSVIEW.c` from the `/IP/` directory is included in the project.

29.3 Enable profiling

Profiling can be included or excluded at compile-time and enabled at run-time. When profiling is excluded, no additional overhead in performance or memory usage is generated. Even when profiling is enabled the overhead is minimal, due to the efficient implementation of SystemView.

To include profiling, define `IP_SUPPORT_PROFILE` as 1 in the embOS/IP configuration (`IP_Conf.h`) or in the project preprocessor defines.

Per default profiling is included when the global define `SUPPORT_PROFILE` is set.

Profiling the end of function calls may be enabled or disabled separately with the define `IP_SUPPORT_PROFILE_END_CALL` . When profiling is enabled it may be defined as 0 to disable recording end of function calls and therefore save bandwidth.

```
#if defined(SUPPORT_PROFILE) && (SUPPORT_PROFILE)
    #ifndef IP_SUPPORT_PROFILE
        #define IP_SUPPORT_PROFILE 1
    #endif
#endif

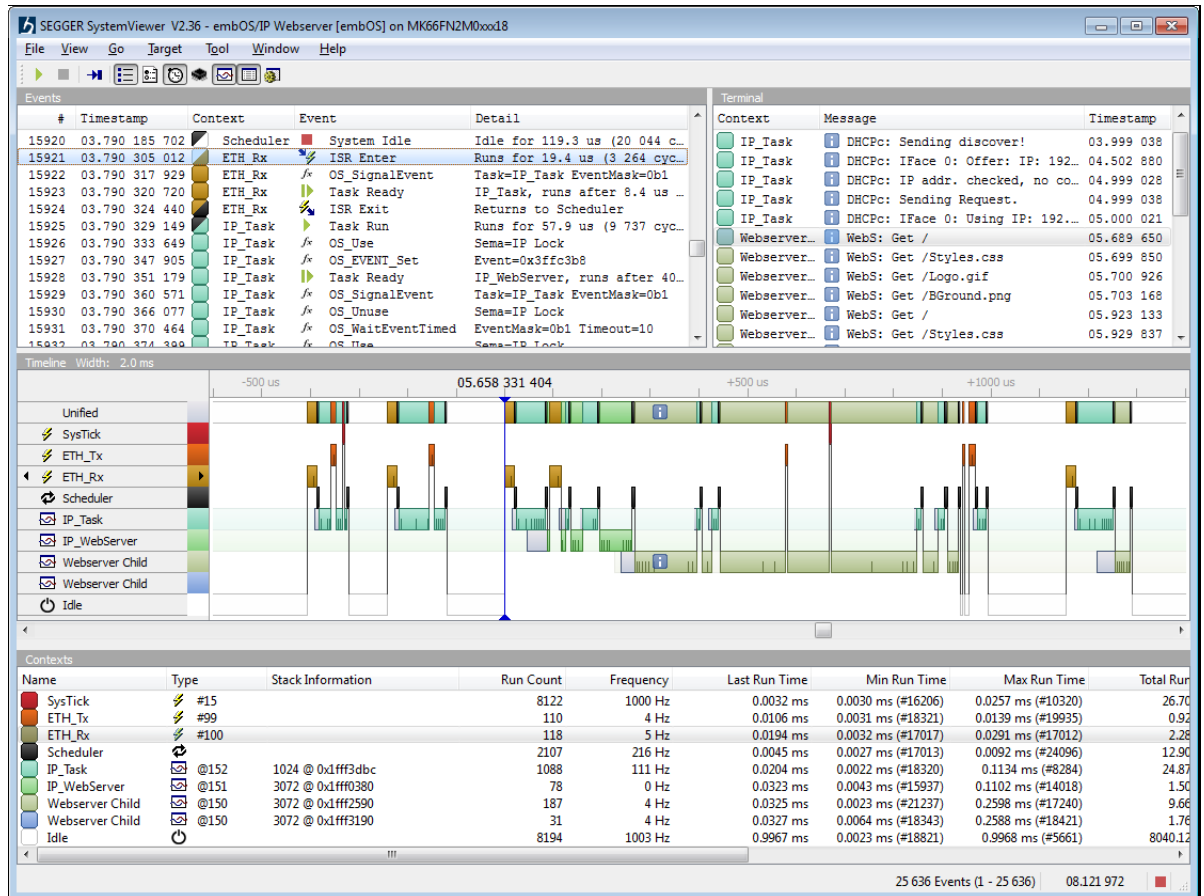
#if defined(IP_SUPPORT_PROFILE)
    #ifndef IP_SUPPORT_PROFILE_END_CALL
        #define IP_SUPPORT_PROFILE_END_CALL IP_SUPPORT_PROFILE
    #endif
#else
    #ifndef IP_SUPPORT_PROFILE_END_CALL
        #define IP_SUPPORT_PROFILE_END_CALL 0
    #endif
#endif
```

To enable profiling at run-time, `IP_SYSVIEW_Init()` needs to be called. Profiling can be enabled at any time, it is recommended to do this in the user-provided configuration `IP_X_Config()` :

```
/* *****
 *
 *      IP_X_Config()
 */
void IP_X_Config(void) {
    #if IP_SUPPORT_PROFILE
        IP_SYSVIEW_Init();
    #endif
    ...
}
```


29.4 Recording and analysing profiling information

When profiling is included and enabled embOS/IP generates profiling events. On a system which supports RTT (i.e. ARM Cortex-M and Renesas RX) the data can be read and analysed with SystemView and a J-Link. Connect the J-Link to the target system using the default debug interface and start the SystemView host application. If the system does not support RTT, SystemView can be configured for single-shot or post-mortem mode. Please refer to the SystemView User Manual for more information.



Chapter 30

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.

30.1 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.

30.2 Testing stability

embOS/IP allows to define drop-rates for both receiver and 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.

30.3 API functions

Function	Description
Filter functions	
<code>IP_Log()</code>	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.
<code>IP_Warn()</code>	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.
<code>IP_AddLogFilter()</code>	Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.
<code>IP_AddWarnFilter()</code>	Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.
<code>IP_SetLogFilter()</code>	Sets the mask that defines which logging message should be displayed.
<code>IP_SetWarnFilter()</code>	Sets the mask that defines which warning message should be displayed.
General debug functions/macros	
<code>IP_PANIC()</code>	Called if the stack encounters a critical situation.

Table 30.1: embOS/IP debugging API function overview

30.3.1 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
<code>FilterMask</code>	Specifies which logging messages should be added to the filter mask. Refer to <i>Message types</i> on page 804 for a list of valid values for parameter <code>FilterMask</code> .

Table 30.2: 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
```

30.3.2 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
<code>FilterMask</code>	Specifies which warning messages should be added to the filter mask. Refer to <i>Message types</i> on page 804 for a list of valid values for parameter <code>FilterMask</code> .

Table 30.3: 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
```


30.3.3 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
<code>FilterMask</code>	Specifies which logging messages should be displayed. Refer to <i>Message types</i> on page 804 for a list of valid values for parameter <code>FilterMask</code> .

Table 30.4: 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 422 for more information.

30.3.4 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 <i>Message types</i> on page 804 for a list of valid values for parameter FilterMask .

Table 30.5: 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 422 for more information.

30.3.5 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.

30.4 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 801 and *IP_SetWarnFilter()* on page 802 as well as *IP_AddLogFilter()* on page 799 and *IP_AddWarnFilter()* on page 799 for more information about using the message types.

Symbolic name	Description
<code>IP_MTYPE_INIT</code>	Activates output of messages from the initialization of the stack that should be logged.
<code>IP_MTYPE_CORE</code>	Activates output of messages from the core of the stack that should be logged.
<code>IP_MTYPE_ALLOC</code>	Activates output of messages from the memory allocating module of the stack that should be logged.
<code>IP_MTYPE_DRIVER</code>	Activates output of messages from the driver that should be logged.
<code>IP_MTYPE_ARP</code>	Activates output of messages from ARP module that should be logged.
<code>IP_MTYPE_IP</code>	Activates output of messages from IP module that should be logged.
<code>IP_MTYPE_TCP_CLOSE</code>	Activates output of messages from TCP module that should be logged when a TCP connection gets closed.
<code>IP_MTYPE_TCP_OPEN</code>	Activates output of messages from TCP module that should be logged when a TCP connection gets opened.
<code>IP_MTYPE_TCP_IN</code>	Activates output of messages from TCP module that should be logged if a TCP packet is received.
<code>IP_MTYPE_TCP_OUT</code>	Activates output of messages from TCP module that should be logged if a TCP packet is sent.
<code>IP_MTYPE_TCP_RTT</code>	Activates output of messages from TCP module regarding TCP roundtrip time.
<code>IP_MTYPE_TCP_RXWIN</code>	Activates output of messages from TCP module regarding peer TCP Rx window size.
<code>IP_MTYPE_TCP</code>	Activates output of messages from TCP that module should be logged.
<code>IP_MTYPE_UDP_IN</code>	Activates output of messages from UDP module that should be logged when a UDP packet is received.
<code>IP_MTYPE_UDP_OUT</code>	Activates output of messages from UDP module that should be logged if a UDP packet is sent.
<code>IP_MTYPE_UDP</code>	Activates output of messages from UDP module that should be logged if a UDP packet is sent or received.
<code>IP_MTYPE_LINK_CHANGE</code>	Activates output of messages regarding to the link change process.
<code>IP_MTYPE_AUTOIP</code>	Activates output of from the AutoIP module that should be logged.
<code>IP_MTYPE_DHCP</code>	Activates output of messages from DHCP client module that should be logged.
<code>IP_MTYPE_DHCP_EXT</code>	Activates output of optional messages from DHCP client module that should be logged.

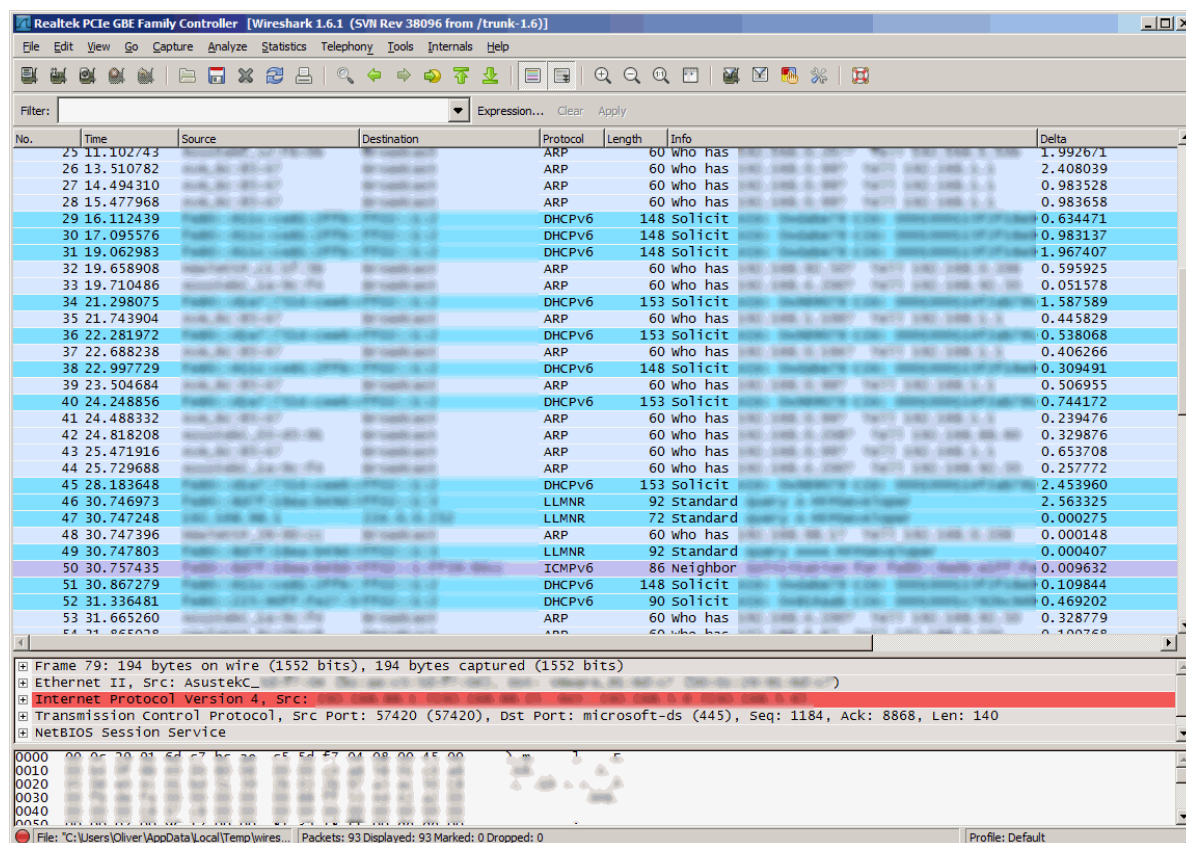
Table 30.6: embOS/IP message types

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 30.6: embOS/IP message types

30.5 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 31

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.

31.1 General information

The complexity of the IP/OS Interface depends on the task model selected. Refer to *Tasks and interrupt usage* on page 25 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.

31.2 OS layer API functions

Function	Description
General macros	
<code>IP_OS_Delay()</code>	Blocks the calling task for a given time.
<code>IP_OS_DisableInterrupt()</code>	Disables interrupts.
<code>IP_OS_EnableInterrupt()</code>	Enables interrupts.
<code>IP_OS_GetTime32()</code>	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.
<code>IP_OS_Init()</code>	Creates and initializes all objects required for task synchronization. These are 2 events (for <code>IP_Task</code> and <code>IP_RxTask</code>) and one semaphore for protection of critical code which may not be executed from multiple task at the same time.
<code>IP_OS_Lock()</code>	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.
<code>IP_OS_Unlock()</code>	Unlocks the single lock used locked by a previous call to <code>IP_OS_Lock()</code> .
IP_Task synchronization	
<code>IP_OS_SignalNetEvent()</code>	Wakes the <code>IP_Task</code> if it is waiting for a NET-event or timeout in the function <code>IP_OS_WaitNetEvent()</code> .
<code>IP_OS_WaitNetEvent()</code>	Called from <code>IP_Task</code> only. Blocks until the timeout expires or a NET-event occurs, meaning <code>IP_OS_SignalNetEvent()</code> is called from an other task or ISR.
IP_RxTask synchronization	
<code>IP_OS_SignalRxEvent()</code>	Wakes the <code>IP_RxTask</code> if it is waiting for a NET-event or timeout in the function <code>IP_OS_WaitRxEvent()</code> .
<code>IP_OS_WaitRxEvent()</code>	Optional. Called from <code>IP_RxTask</code> , if it is used to receive data. Blocks until the timeout expires or a NET-event occurs, meaning <code>IP_OS_SignalRxEvent()</code> is called from the ISR.
Application task synchronization	
<code>IP_OS_WaitItem()</code>	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.
<code>IP_OS_WaitItemTimed()</code>	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.
<code>IP_OS_SignalItem()</code>	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 31.1: Target OS interface function list

31.2.1 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 32

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.

32.1 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.

32.1.1 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 32.1: ARM7 sample configuration

32.1.1.1 ROM usage

The following table shows the ROM requirement of embOS/IP:

Description	ROM
embOS/IP - complete stack	approximately 19.0Kbytes

The memory requirements of a interface driver is about 1.5 - 2.0Kbytes.

32.1.1.2 RAM usage

The following table shows the RAM requirement of embOS/IP:

Description	RAM
embOS/IP - complete stack w/o buffers	approximately 1.5Kbytes

32.1.2 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 32.2: ARM7 sample configuration

32.1.2.1 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.

32.1.2.2 RAM usage

The following table shows the RAM requirement of embOS/IP:

Description	RAM
embOS/IP - complete stack w/o buffers	approximately 4.5Kbytes

32.2 Performance

32.2.1 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 32.3: 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.

32.2.2 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 32.4: 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 33

Appendix A - File system abstraction layer

33.1 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

33.2 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);
    //
    // Additional operations
    //
    int     (*pfIsFolder)   ( const char * sPath);
    int     (*pfMove)       ( const char * sOldFilename,
                             const char * sNewFilename);
} IP_FS_API;
```

Elements of IP_FS_API

Function	Description
Read only file system functions (required)	
<code>pfOpenFile</code>	Pointer to a function that creates/opens a file and returns the handle of these file.
<code>pfCloseFile</code>	Pointer to a function that closes a file.
<code>pfReadAt</code>	Pointer to a function that reads a file.
<code>pfGetLen</code>	Pointer to a function that returns the length of a file.
Directory query operations	
<code>pfForEachDirEntry</code>	Pointer to a function which is called for each directory entry.
<code>pfGetDirEntryFileName</code>	Pointer to a function that returns the name of a file entry.
<code>pfGetDirEntryFileSize</code>	Pointer to a function that returns the size of a file.
<code>pfGetDirEntryFileTime</code>	Pointer to a function that returns the time-stamp of a file.
<code>pfGetDirEntryAttributes</code>	Pointer to a function that returns the attributes of a directory entry.
Write file operations	
<code>pfCreate</code>	Pointer to a function that creates a file.
<code>pfDeleteFile</code>	Pointer to a function that deletes a file.
<code>pfRenameFile</code>	Pointer to a function that renames a file.
<code>pfWriteAt</code>	Pointer to a function that writes a file.
Additional directory operations (optional)	
<code>pfMKDir</code>	Pointer to a function that creates a directory.
<code>pfRMDir</code>	Pointer to a function that deletes a directory.
Additional operations (optional)	
<code>pfIsFolder</code>	Pointer to a function that checks if a path is a folder.
<code>pfMove</code>	Pointer to a function that moves a file or directory.

Table 33.1: embOS/IP file system API function overview

33.2.1 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 = {
    //
    // Read only file operations.
    //
    _FS_Open,
    _Close,
    _ReadAt,
    _GetLen,
    //
    // Simple directory operations.
    //
    _ForEachDirEntry,
    _GetDirEntryFileName,
    _GetDirEntryFileSize,
    _GetDirEntryFileTime,
    _GetDirEntryAttributes,
    //
    // Simple write type file operations.
    //
    _Create,
    _DeleteFile,
    _RenameFile,
    _WriteAt,
    //
    // Additional directory operations
    //
    _MKDir,
    _RMDir,
    //
    // Additional operations
    //
    _IsFolder,
    _Move
};
```

The emFile interface is used in all SEGGER Eval Packages.

33.2.2 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 = {
    //
    // Read only file operations.
    //
    _FS_RO_FS_Open,
    _FS_RO_Close,
    _FS_RO_ReadAt,
    _FS_RO_GetLen,
    //
    // Simple directory operations.
    //
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    //
    // Simple write type file operations.
    //
    NULL,
    NULL,
    NULL,
    NULL,
    //
    // Additional directory operations
    //
    NULL,
    NULL,
    //
    // Additional operations
    //
    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.

33.2.3 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.

33.2.4 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  
    //  
    // Additional operations  
    //  
    _IP_FS_WIN32_IsFolder,  
    _IP_FS_WIN32_Move  
};
```

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 34

Support

This chapter should help if any problem occurs, e.g. with the hardware or the use of the embOS/IP functions, and describes how to contact the embOS/IP support.

34.1 Contacting support

If you are a registered embOS/IP user and you need to contact the embOS/IP support, please send the following information via email to **support_embosip@segger.com**:

- The embOS/IP version.
- Your embOS/IP registration number.
- If you are unsure about the above information, you may also use the name of the embOS/IP ZIP-file (which contains the above information).
- A detailed description of the problem.
- Optionally, a project with which we can reproduce the problem.

Chapter 35

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

C

Compile-time configuration 426

D

Debugging

 IP_Panic() 803

DHCP client

 IP_DHCP_Activate() 294

 IP_DHCP_ConfigAlwaysStartInit() 295

 IP_DHCP_ConfigOnActivate() 296

 IP_DHCP_ConfigOnFail() 297

 IP_DHCP_ConfigOnLinkDown() 298

 IP_DHCP_Halt() 300, 302

 IP_DHCP_Renew() 301

 IP_DHCP_SetClientId() 303

E

embOS/IP

 Features 22

 Integrating into your system 36

 layers 23

F

FS abstraction layer

 emFile interface 821

I

IP stack ACD functions

 IP_ACD_Activate() 328

 IP_ACD_Config() 329

IP stack AutoIP functions

 IP_AutoIP_Activate() 320

 IP_AutoIP_Halt() 321

 IP_AutoIP_SetStartIP() 323

 IP_AutoIP_SetUserCallback() 322

IP stack configuration functions

 IP_AddBuffers() 53

 IP_AddEtherInterface() 54

 IP_AddLoopbackInterface() 56

 IP_AddMemory() 57

 IP_AddVirtEtherInterface() 55

 IP_AllowBackpressure() 58

 IP_ARP_CleanCache() 60

 IP_ARP_CleanCacheByInterface() 61

 IP_ARP_ConfigAgeout() 62

 IP_ARP_ConfigAgeoutNoReply() 63

 IP_ARP_ConfigAgeoutSniff() 64

 IP_ARP_ConfigAllowGratuitousARP() 65

 IP_ARP_ConfigMaxPending() 66

 IP_ARP_ConfigMaxRetries() 67

 IP_ARP_ConfigNumEntries() 68

 IP_AssignMemory() 59

 IP_BSP_SetAPI() 69

 IP_ConfigOffCached2Uncached() 74

 IP_ConfigTCPSpace() 75

 IP_DisableIPRxChecksum() 76

 IP_DNS_GetServer() 77

 IP_DNS_GetServerEx() 78

 IP_DNS_SetMaxTTL() 79

 IP_DNS_SetServer() 80

 IP_DNS_SetServerEx() 81

 IP_EnableIPRxChecksum() 82

 IP_GetMaxAvailPacketSize() 83

 IP_GetMTU() 84

 IP_GetPrimaryIFace() 85

 IP_ICMP_Add() 86

 IP_ICMP_DisableRxChecksum() 87

 IP_ICMP_EnableRxChecksum() 88

 IP_IGMP_Add() 89

 IP_IGMP_AddEx() 90

 IP_IGMP_JoinGroup() 91

 IP_IGMP_LeaveGroup() 92

 IP_RAW_Add() 93

 IP_SetAddrMask() 94

 IP_SetAddrMaskEx() 95

 IP_SetGWAddr() 96

 IP_SetHWAddr() 97

 IP_SetHWAddrEx() 98

 IP_SetMTU() 99

 IP_SetPrimaryIFace() 100

 IP_SetSupportedDuplexModes() 101

 IP_SetTTL() 102

 IP_SOCKET_ConfigSelectMultiplicator() 103

 IP_SOCKET_SetDefaultOptions() 104

IP_SOCKET_SetLimit()	105	IP_Init()	119
IP_TCP_Add()	107	IP_RxTask()	121
IP_TCP_DisableRxChecksum()	108	IP_Task()	120
IP_TCP_EnableRxChecksum()	109	IP stack Modem functions	
IP_TCP_Set2MSLDelay()	110	IP_MODEM_Connect()	656
IP_TCP_SetConnKeepaliveOpt()	111	IP_MODEM_Disconnect()	657
IP_TCP_SetRetransDelayRange()	112	IP_MODEM_GetResponse()	658
IP_UDP_Add()	113	IP_MODEM_SendString()	659
IP_UDP_AddEchoServer()	114	IP_MODEM_SendStringEx()	660
IP_UDP_DisableRxChecksum()	115	IP_MODEM_SetAuthInfo()	662
IP_UDP_EnableRxChecksum()	116	IP_MODEM_SetConnectTimeout()	663
Structure IP_BSP_API	203	IP_MODEM_SetInitCallback()	664
Structure IP_BSP_INSTALL_ISR_PARA	202	IP_MODEM_SetInitString()	665
Structure SEGGER_CACHE_CONFIG	204	IP_MODEM_SetSwitchToCmdDelay()	666
IP stack DHCP server functions		IP stack NetBIOS functions	
IP_DHCP_ConfigDNSAddr()	308	IP_NETBIOS_Init()	681
IP_DHCP_ConfigGWAddr()	309	IP_NETBIOS_Start()	682
IP_DHCP_ConfigMaxLeaseTime()	310	IP_NETBIOS_Stop()	683
IP_DHCP_ConfigPool()	311	IP stack network configuration functions	
IP_DHCP_Halt()	312	IP_NI_ConfigPoll()	124
IP_DHCP_Init()	313	IP_NI_ForceCaps()	125
IP_DHCP_Start()	314	IP_NI_SetTxBufferSize()	126
IP stack functions		IP stack PHY configuration functions	
IP_AddAfterInitHook()	157	IP_NI_ConfigPHYAddr()	128
IP_AddEtherTypeHook()	158	IP_NI_ConfigPHYMode()	129
IP_AddOnPacketFreeHook()	160	IP_PHY_AddDriver()	130
IP_AddStateChangeHook()	161	IP_PHY_AddResetHook()	132
IP_Alloc()	162	IP_PHY_ConfigAddr()	133
IP_AllocEtherPacket()	163	IP_PHY_ConfigAltAddr()	134
IP_AllocEx()	164	IP_PHY_ConfigSupportedModes()	135
IP_Connect()	165	IP_PHY_ConfigUseStaticFilters()	136
IP_Disconnect()	166	IP_PHY_DisableCheck()	137
IP_Err2Str()	167	IP_PHY_DisableCheckEx()	138
IP_FindIFaceByIP()	168	IP_PHY_ReInit()	139
IP_Free()	169	IP_PHY_SetWdTimeout()	140
IP_FreePacket()	170	IP stack PPP functions	
IP_GetAddrMask()	171	IP_PPP_AddInterface()	650
IP_GetCurrentLinkSpeed()	172	IP stack PPPoE functions	
IP_GetCurrentLinkSpeedEx()	173	IP_PPPOE_AddInterface()	644
IP_GetFreePacketCnt()	174	IP_PPPOE_ConfigRetries()	645
IP_GetIFaceHeaderSize()	175	IP_PPPOE_Reset()	646
IP_GetIPAddr()	176–178	IP_PPPOE_SetAuthInfo()	647
IP_GetIPPacketInfo()	179	IP_PPPOE_SetUserCallback()	648
IP_GetRawPacketInfo()	180	IP stack SNMP agent functions	
IP_GetVersion()	181	IP_SNMP_AGENT_AddCommunity()	723
IP_ICMP_SetRxHook()	182	IP_SNMP_AGENT_AddInformReponseHook()	725
IP_IFaceIsReady()	183	IP_SNMP_AGENT_AddMIB()	724
IP_IFaceIsReadyEx()	184	IP_SNMP_AGENT_AddMIB_IsoOrgDodIntern	
IP_IsExpired()	185	etIetfMib2Interfaces()	738
IP_NI_GetAdminState()	186	IP_SNMP_AGENT_AddMIB_IsoOrgDodIntern	
IP_NI_GetIFaceType()	187	etIetfMib2System()	739
IP_NI_GetState()	188	IP_SNMP_AGENT_AddMIB_IsoOrgDodIntern	
IP_NI_GetTxQueueLen()	190	etPrivateEnterprise()	740
IP_NI_SetAdminState()	189	IP_SNMP_AGENT_CancelInform()	726
IP_PrintIPAddr()	191	IP_SNMP_AGENT_CheckInformStatus()	727
IP_ResolveHost()	192	IP_SNMP_AGENT_CloseVarbind()	741
IP_SendEtherPacket()	193	IP_SNMP_AGENT_DecodeOIDValue()	781
IP_SendPacket()	194	IP_SNMP_AGENT_DeInit()	728
IP_SendPing()	195	IP_SNMP_AGENT_EncodeOIDValue()	782
IP_SendPingEx()	196	IP_SNMP_AGENT_Exec()	729
IP_SetIFaceConnectHook()	197	IP_SNMP_AGENT_GetMessageType()	730
IP_SetIFaceDisconnectHook()	198	IP_SNMP_AGENT_Init()	731
IP_SetPacketToS()	199	IP_SNMP_AGENT_OpenVarbind()	742
IP_SetRxHook()	200	IP_SNMP_AGENT_ParseBits()	763
IP stack management functions		IP_SNMP_AGENT_ParseCounter()	764
IP_DeInit()	118	IP_SNMP_AGENT_ParseCounter32()	765
IP_Exec()	122		

- IP_SNMP_AGENT_ParseCounter64() ... 766
- IP_SNMP_AGENT_ParseDouble() 767
- IP_SNMP_AGENT_ParseFloat() 768
- IP_SNMP_AGENT_ParseGauge() 769
- IP_SNMP_AGENT_ParseGauge32() 770
- IP_SNMP_AGENT_ParseInteger() 771
- IP_SNMP_AGENT_ParseInteger32() ... 772
- IP_SNMP_AGENT_ParseInteger64() ... 773
- IP_SNMP_AGENT_ParseIpAddress() ... 774
- IP_SNMP_AGENT_ParseOctetString() . 775
- IP_SNMP_AGENT_ParseOID() 776
- IP_SNMP_AGENT_ParseOpaque() 777
- IP_SNMP_AGENT_ParseTimeTicks() 778
- IP_SNMP_AGENT_ParseUnsigned32() . 779
- IP_SNMP_AGENT_ParseUnsigned64() . 780
- IP_SNMP_AGENT_PrepareTrapInform() 732
- IP_SNMP_AGENT_ProcessInformResponse()
..... 734
- IP_SNMP_AGENT_ProcessRequest() ... 735
- IP_SNMP_AGENT_SendTrapInform() .. 736
- IP_SNMP_AGENT_SetCommunityPerm() ..
737
- IP_SNMP_AGENT_StoreBits() 743
- IP_SNMP_AGENT_StoreCounter() 744
- IP_SNMP_AGENT_StoreCounter32() ... 745
- IP_SNMP_AGENT_StoreCounter64() ... 746
- IP_SNMP_AGENT_StoreCurrentMibOidAndIn
dex() 747
- IP_SNMP_AGENT_StoreDouble() 748
- IP_SNMP_AGENT_StoreFloat() 749
- IP_SNMP_AGENT_StoreGauge() 750
- IP_SNMP_AGENT_StoreGauge32() 751
- IP_SNMP_AGENT_StoreInstanceNA() . 752
- IP_SNMP_AGENT_StoreInteger() 753
- IP_SNMP_AGENT_StoreInteger32() ... 754
- IP_SNMP_AGENT_StoreInteger64() ... 755
- IP_SNMP_AGENT_StoreIpAddress() ... 756
- IP_SNMP_AGENT_StoreOctetString() . 757
- IP_SNMP_AGENT_StoreOID() 758
- IP_SNMP_AGENT_StoreOpaque() 759
- IP_SNMP_AGENT_StoreTimeTicks() 760
- IP_SNMP_AGENT_StoreUnsigned32() . 761
- IP_SNMP_AGENT_StoreUnsigned64() . 762
- SNMP agent data structures 783
- Structure IP_SNMP_AGENT_API 783
- Structure
 IP_SNMP_AGENT_MIB2_INTERFACES_A
 PI 786
- Structure
 IP_SNMP_AGENT_MIB2_SYSTEM_API
 785
- Structure IP_SNMP_AGENT_PERM 784
- IP stack SNTTP client functions
- IP_SNTTP_ConfigTimeout() 694
- IP_SNTTP_GetTimeStampFromServer() 695
- IP stack stack functions
- Data-structures 201
- IP stack statistics functions
- IP_STATS_EnableIFaceCounters() 142
- IP_STATS_GetIFaceCounters() 143
- IP_STATS_GetLastLinkStateChange() . 144
- IP_STATS_GetRxBytesCnt() 145
- IP_STATS_GetRxDiscardCnt() 146
- IP_STATS_GetRxErrCnt() 147
- IP_STATS_GetRxNotUnicastCnt() 148
- IP_STATS_GetRxUnicastCnt() 149
- IP_STATS_GetRxUnknownProtoCnt() .. 150
- IP_STATS_GetTxBytesCnt() 151
- IP_STATS_GetTxDiscardCnt() 152
- IP_STATS_GetTxErrCnt() 153
- IP_STATS_GetTxNotUnicastCnt() 154
- IP_STATS_GetTxUnicastCnt() 155
- Structure IP_STATS_IFACE 205
- IP stack Tail Tagging functions
- IP_MICREL_TAIL_TAGGING_AddInterface()
 365
- IP stack UPnP functions
- IP_UPNP_Activate() 346
- IP stack VLAN functions
- IP_VLAN_AddInterface() 354
- IP stack Web server functions
- Callback IP_WEBS_pfMethod 559
- Callback IP_WEBS_pfRequestNotify 561
- IP_UTIL_BASE64_Decode() 545
- IP_UTIL_BASE64_Encode() 546
- IP_WEBS_AddFileTypeHook() 495
- IP_WEBS_AddUpload() 494
- IP_WEBS_AddVFileHook() 525
- IP_WEBS_CompareFileNameExt() 528
- IP_WEBS_ConfigBufSizes() 498
- IP_WEBS_ConfigRootPath() 499
- IP_WEBS_ConfigSendVFileHeader() ... 529
- IP_WEBS_ConfigSendVFileHookHeader() .
 530
- IP_WEBS_ConfigUploadRootPath() 500
- IP_WEBS_DecodeAndCopyStr() 531
- IP_WEBS_DecodeString() 532
- IP_WEBS_Flush() 501
- IP_WEBS_GetConnectInfo() 538
- IP_WEBS_GetDecodedStrLen() 533
- IP_WEBS_GetNumParas() 534
- IP_WEBS_GetParaValue() 535
- IP_WEBS_GetParaValuePtr() 536
- IP_WEBS_GetURI() 539
- IP_WEBS_Init() 502
- IP_WEBS_METHOD_AddHook() 541
- IP_WEBS_METHOD_CopyData() 544
- IP_WEBS_OnConnectionLimit() 506
- IP_WEBS_Process() 507
- IP_WEBS_ProcessEx() 508
- IP_WEBS_ProcessLast() 509
- IP_WEBS_ProcessLastEx() 510
- IP_WEBS_Redirect() 511
- IP_WEBS_Reset() 512
- IP_WEBS_RetrieveUserContext() 513
- IP_WEBS_SendHeader() 514
- IP_WEBS_SendHeaderEx() 515
- IP_WEBS_SendLocationHeader() 516
- IP_WEBS_SendMem() 517
- IP_WEBS_SendString() 518
- IP_WEBS_SendStringEnc() 519
- IP_WEBS_SendUnsigned() 520
- IP_WEBS_SetFileInfoCallback() 521
- IP_WEBS_SetHeaderCacheControl() ... 522
- IP_WEBS_StoreUserContext() 523
- IP_WEBS_UseRawEncoding() 540
- Structure IP_WEBS_FILE_INFO 550
- Structure WEBS_ACCESS_CONTROL .. 548
- Structure WEBS_APPLICATION 549
- Structure WEBS_CGI 547
- Structure WEBS_FILE_TYPE 553
- Structure WEBS_FILE_TYPE_HOOK 554
- Structure WEBS_METHOD_HOOK 555

Structure WEBS_REQUEST_NOTIFY_HOOK
557
Structure WEBS_VFILE_APPLICATION 551
Structure WEBS_VFILE_HOOK 552
Web server data structures 547

L

Logging functions
IP_AddLogFilter() 799
IP_AddWarnFilter() 800
IP_SetLogFilter() 801
IP_SetWarnFilter() 802

N

Network interface drivers
ATMEL AT91SAM7X 383
ATMEL AT91SAM9260 387
DAVICOM DM9000 390
FREESCALE ColdFire MCF5329 393
NXP LPC23xx/24xx 398
ST STR912 400

O

OS integration 807
API functions 809

P

PHY driver
Generic driver API functions
IP_PHY_GENERIC_AddResetHook()
413
IP_PHY_GENERIC_RemapAccess()
414
Micrel Switch PHY driver API functions
IP_PHY_MICREL_SWITCH_ConfigLe
arnDisable() 417
IP_PHY_MICREL_SWITCH_ConfigPo
rtNumber() 416
IP_PHY_MICREL_SWITCH_ConfigR
xEnable() 418
IP_PHY_MICREL_SWITCH_ConfigTa
ilTagging() 419
IP_PHY_MICREL_SWITCH_ConfigT
xEnable() 420

R

RAW zero-copy
IP_RAW_Alloc() 279
IP_RAW_Close() 280
IP_RAW_Free() 281
IP_RAW_GetDataPtr() 282
IP_RAW_GetDataSize() 283
IP_RAW_GetDestAddr() 284
IP_RAW_GetIFIndex() 285
IP_RAW_GetSrcAddr() 286
IP_RAW_Open() 287
IP_RAW_Send() 288
IP_RAW_SendAndFree() 289

S

Socket functions
accept() 209
bind() 211
closesocket() 212
connect() 213
gethostbyname() 215
getpeername() 217
getsockname() 218
IP_SOCKET_GetAddrFam() 236
IP_SOCKET_GetLocalPort() 237
listen() 223
recv() 224
recvfrom() 225
select() 226
send() 229
sendto() 230
setsockopt() 231
socket() 234
Structure hostent 241
Structure in_addr 240
Structure sockaddr 238
Structure sockaddr_in 239
Syntax, conventions used 11

T

TCP zero-copy
IP_TCP_Alloc() 248
IP_TCP_AllocEx() 249
IP_TCP_Free() 250
IP_TCP_Send() 251
IP_TCP_SendAndFree() 252
TFTP
IP_TFTP_InitContext() 631
IP_TFTP_RecvFile() 632
IP_TFTP_SendFile() 633
IP_TFTP_ServerTask() 634

U

UDP zero-copy
IP_UDP_Alloc() 258
IP_UDP_Close() 259
IP_UDP_FindFreePort() 260
IP_UDP_Free() 261
IP_UDP_GetDataPtr() 263
IP_UDP_GetDataSize() 262
IP_UDP_GetDestAddr() 264
IP_UDP_GetFPort() 265
IP_UDP_GetIFIndex() 266
IP_UDP_GetLPort() 267
IP_UDP_Open() 269
IP_UDP_OpenEx() 270
IP_UDP_Send() 271
IP_UDP_SendAndFree() 272
Utility functions
IP_UTIL_BASE64_Decode() 545
IP_UTIL_BASE64_Encode() 546