emNet

CPU independent TCP/IP stack
for embedded applications

User Guide & Reference Manual

Document: UM07001
Software Version: 3.54.1
Revision: 0
Date: December 11, 2023

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

Trademarks

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

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

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	ticket_emnet@segger.com*
Internet:	www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: December 11, 2023

Software	Revision	Date	By	Description
3.54.1	0	231211	YR	Update to latest software version.
3.54.0	0	231204	YR	Chapter "File system abstraction layer" updated.
3.52.0	0	230928	YR	Update to latest software version.
3.50.5	1	230706	OO	Chapter "UPnP (Add-on)" updated. • Removed link to old resource and replaced it with plain text.
3.50.5	0	230627	OO	Chapter "Address Collision Detection (ACD)" updated. • Added information regarding *EtherNet/IP usage* on page 502. • `IP_ACD_EndAnnounce()` added. • `IP_ACD_UpdateBackgroundPeriod()` added. • New member `InitState` added to structure `IP_ACD_EX_CONFIG` . Chapter "Internet Protocol version 6 (IPv6) (Add-on)" updated. • `IP_IPV6_ResolveHost()` added. Chapter "emFTP server (Add-on)" updated. • `IP_FTPS_UseRenameToFullPath()` added.
3.50.3	0	230602	OO	Fixed some images. Chapter "Socket interface" updated. • `IP_ERR_NO_MEM` removed from possible error codes as it is replaced with the existing `IP_ERR_NOMEM` .
3.50.2	0	230327	OO	Chapter "Socket interface" updated. • sogetopt() `SO_ERROR` error code `IP_ERR_USER_ABORT` added. • `IP_SOCKET_AbortRead()` added. • Description for `select()` updated.
3.50.0	1	230217	OO	Chapter "Performance & resource usage" updated. • Corrected RAM footprint for Cortex-M3.
3.50.0	0	230215	OO	Chapter "OS integration" updated. • Renamed/updated `IP_OS_WaitNetEvent()` to `IP_OS_WaitNetEvent-Timed()` . • Renamed/updated `IP_OS_WaitRxEvent()` to `IP_OS_WaitRxEvent-Timed()` . • `IP_OS_WaitItem()` removed. • `IP_OS_SignalDTaskEvent()` added. • `IP_OS_WaitDTaskEventTimed()` added. Chapter "Core functions" updated. • Description for `IP_Task()` updated. • Description for `IP_RxTask()` updated. • `IP_TASK_Init()` added. • `IP_TASK_Exec()` added. • `IP_TASK_WaitForEvent()` added. • `IP_RXTASK_Init()` added. • `IP_RXTASK_Exec()` added. • `IP_RXTASK_WaitForEvent()` added. • `IP_PHY_ReadReg()` added. • `IP_PHY_WriteReg()` added. • New PHY mode flag `IP_PHY_MODE_NO_AUTONEG` added. • Description for `IP_SetSupportedDuplexModes()` updated. • Description for `IP_PHY_ConfigSupportedModes()` updated. • Description for `IP_TCP_DisableRxChecksum()` updated. • Description for `IP_TCP_EnableRxChecksum()` updated. Chapter "WiFi support" updated. • Description for `IP_DTASK_Task()` updated. • Description for `IP_DTASK_Exec()` updated. • `IP_DTASK_Init()` added. • `IP_DTASK_ExecAll()` added. • `IP_DTASK_WaitForEvent()` added. • `IP_DTASK_ConfigTimeout()` renamed to `IP_DTASK_SetTimeout()`. • `IP_DTASK_GetTimeout()` added.
3.42.10	0	221212	OO/ MH	Chapter "Socket interface" updated. • Updated available socket options for `getsockopt()`/`setsockopt()` .

Software	Revision	Date	By	Description
3.42.7	0	221014	OO	Chapter “Core functions” updated. • IP_GetMemPoolInfo() added. • Description for IP_SetMTU() updated (can now also increase the value after the configuration phase). • Description for IP_DNS_GetServer() return value updated (return value is actually in host endianness). • Description for IP_DNS_GetServerEx() return-parameter “pAddr” updated (added endianness information). • IP_IGMP_ConfigV2AlwaysReport() added. • IP_IGMP_JoinGroup_AutoRejoin() added. • Descriptions for IP_IGMP_JoinGroup() and IP_IGMP_LeaveGroup() updated. Chapter “DHCP Client” updated. • IP_DHCPC_ConfigRequestLeaseTime() added. Chapter “PTP Ordinary Clock (Add-on)” updated. • IP_PTP_MASTER_Config() added along with its used structures. Chapter “SNMP agent (Add-on)” updated. • Added API, structures and information for SNMPv3 support. Chapter “Socket interface” updated. • IP_SOCKET_ConfigSelectMultiplicator() added. • IP_SOCKET_GetNumRxBytes() added. • IP_SOCKET_SetDefaultOptions() added. • IP_SOCKET_SetLimit() added. • IP_SOCKET_SetLinger() added. • IP_SOCKET_SetRxTimeout() added.
3.42.5	0	220419	OO	Chapter “Core functions” updated. • IP_Shutdown() added.
3.42.4	0	220401	OO	Chapter “Core functions” updated. • IP_ICMP_AddRxHook() added along with its used structures. • IP_ICMP_RemoveRxHook() added. Chapter “Internet Protocol version 6 (IPv6) (Add-on)” updated. • IP_IPV6_GetIPPacketInfo() added. • IP_ICMPV6_AddRxHook() added along with its used structures. • IP_ICMPV6_RemoveRxHook() added.
3.42.3	0	220324	OO	Chapter “PHY drivers” updated. • IP_PHY_MICREL_SWITCH_ConfigUseInternalRmiiClock() added. Chapter “Socket interface” updated. • Non-blocking return value in additional information for recv() and recvfrom() corrected.
3.42.0	0	211123	OO/ YR	Use of new version number scheme. Chapter “Core functions” updated. • IP_ConfigDoNotAddLowLevelChecks_ARP() added. • IP_ConfigDoNotAddLowLevelChecks_UDP() added. • IP_ARP_ConfigAnnounceStaticIP() added. • IP_RemoveEtherTypeHook() added. • Corrected wrong default TTL value in IP_SetLocalMcTTL() description. Chapter “Address Collision Detection (ACD)” updated. • IP_ACD_Halt() added (previously forgotten). • IP_ACD_ActivateEx() added along with its used structures. Chapter “DHCP Client” updated. • IP_DHCPC_AssignCurrentConfig() added. • IP_DHCPC_ConfigAssignConfigManually() added. • IP_DHCPC_ConfigDisableARPCheck() added. • IP_DHCPC_SendDeclineAndHalt() added. • IP_DHCPC_SendDeclineAndResetIP() added. Chapter “MQTT client (Add-on)” updated. • Added MQTT 5 support. Chapter “emFTP client (Add-on)” updated. • Added support for APPE(nd) command. • IP_FTPC_ExecCmdEx() added. • Structure IP_FTPC_CMD_CONFIG added. • FTPC_CMD_LIST sPara can now be used to specify a path to list. Chapter “CoAP client/server (Add-on)” updated. • Configuration switches explanation for IP_COAP_ACK_TIMEOUT . • Configuration switches define IP_COAP_OBS_FORCE_CON_TIMEOUT added. Chapter “NTP client (Add-on)” updated. • IP_NTP_CLIENT_ResetAll() added. Chapter “PPP / PPPoE (Add-on)” updated. • IP_MODEM_SetUartConfig() added.

Software	Revision	Date	By	Description
				• IP_PPP_CHAP_AddWithMD5() added. Chapter "PTP Ordinary Clock (Add-on)" updated. • Information regarding PTP master added. • New API for master and slave added. Chapter "UDP zero-copy interface" updated. • Previously forgotten IP_UDP_AllocEx() added.
3.40e	0	210826	OO	Chapter "Core functions" updated. • IP_PHY_ConfigAfterResetDelay() added. • IP_SetOnIFaceSelectCallback() added. • Structure IP_ON_IFACE_SELECT_INFO added. • Callback IP_ON_IFACE_SELECT_FUNC added. Chapter "Configuring emNet" updated. • Define IP_PHY_AFTER_RESET_DELAY added. Chapter "CoAP client/server (Add-on)" updated. • IP_COAP_SERVER_ConfigClear() was mistakenly listed as IP_COAP_SERVER_ConfigUnset() and was empty. Chapter "TCP Zero-Copy" updated. • Some old defines such as ESHUTDOWN instead of IP_ERR_SHUTDOWN were listed. Chapter "SNTP client (Add-on)" updated. • IP_SNTPC_ConfigAcceptNoSyncSource() added.
3.40a	0	200629	OO	Chapter "DHCP server (Add-on)" updated. • IP_DHCPC_SetVendorOptionsCallback() renamed to IP_DHCPS_SetVendorOptionsCallback() as it has been wrongly given the DHCP client prefix. Chapter "emFTP server (Add-on)" updated. • Structure FTPS_SEND_SIGN_ON_MSG_FUNC added. • IP_FTPS_SetSignOnMsgCallback() added. • IP_FTPS_SendFormattedString() added. • IP_FTPS_SendMem() added. • IP_FTPS_SendString() added. • IP_FTPS_SendUnsigned() added. Chapter "SNMP agent (Add-on)" updated. • Updated *SNMP agent requirements* on page with information about how to transform your decimal PEN into byte BER format. • Added IP_SNMP_GENERIC_TRAP_OID_ENTERPRISE_SPECIFIC OID trap define to IP_SNMP_AGENT_PrepareTrapInform() description. Chapter "Socket interface" updated. • IP_SOCK_recvfrom_info() added.
3.40	0	200402	OO	Manual. • Product name changed from embOS/IP to emNet. • Chapter emWeb server removed from emNet User Guide & Reference Manual. All emWeb related information can be found in the 'emWeb User Guide & Reference Manual'. Chapter "Configuring emNet" updated. • Information for IP_TCP_SetConnKeepaliveOpt() parameters updated. • Information for IP_TCP_SetRetransDelayRange() parameters updated. • Configuration switches regarding TCP retransmits and keepalives added. Chapter "Core functions" updated. • IP_SetNanosecondsCallback() added. • IP_MDNS_ResolveHostSingleIP() added. • IP_NI_ConfigUsePromiscuousMode() added. • IP_NI_PauseRx() added. Chapter "DHCP server (Add-on)" updated. • IP_DHCPS_SetVendorOptionsCallback() added. Chapter "FTP server (Add-on)" updated. • New FTPS_BUFFER_SIZES structure member NumBytesInBufBeforeFlush added. Chapter "PTP Ordinary Clock (Add-on)" updated. • IP_PTP_OC_SetInfoCallback() added. • Structure IP_PTP_INFO added. • Structure IP_PTP_CORRECTION_INFO added. • Structure IP_PTP_OFFSET_INFO added. • Structure IP_PTP_MASTER_INFO added. Chapter "WebSocket (Add-on)" updated. • Added more additional information to IP_WEBSOCKET_Recv() regarding CLOSE frames and their optional application data. Chapter "WiFi support" updated. • Extended the additional information of IP_WIFI_Scan() to cover scan during connect.

Software	Revision	Date	By	Description
				Chapter "WiFi drivers" updated. • IP_NI_WIFI_REDPINE_RS9113_SetUpdateCallback() added.
3.30c	0	190110	OO	Chapter "Configuring emNet" updated. • Additional information for TCP window size configuration added. • Configuration switch IP_SUPPORT_TCP_DELAYED_ACK added.
3.30b	0	181026	OO	Chapter "Core functions" updated. • Additional information for IP_SendEtherPacket() updated. Chapter "PHY drivers" updated. • Information for Microchip/Micrel KSZ8863 added. Chapter "PTP Ordinary Clock (Add-on)" updated. • Information about TAI time representation added. Chapter "RAW zero-copy interface" updated. • IP_RAW_ReducePayloadLen() added. Chapter "UDP zero-copy interface" updated. • IP_UDP_ReducePayloadLen() added. Chapter "Web server (Add-on)" updated. • Progress status WEBS_PROGRESS_STATUS_METHOD_URI_VER_PARSED added to structure WEBS_PROGRESS_INFO . • Added missing "paVFiles" member to WEBS_APPLICATION structure. Chapter "WebSocket (Add-on)" updated. • Link to IP_WEBS_WEBSOCKET_AddHook() added for better explanation of samples. Chapter "WiFi support" updated. • IP_WIFI_SECURITY_WPA_WPA2_MIXED added to IP_WIFI_CONNEC-T_PARAMS description.
3.30	1	180709	OO	Chapter "Internet Protocol version 6 (IPv6) (Add-on)" updated. • IP_IPV6_GetIPv6Addr() added.
3.30	0	180704	OO	Chapter "Core functions" updated. • IP_DisableIPv4() added. • Additional information for IP_AddEtherInterface() added. • Default values for IP_ARP_Config*() added/corrected. Chapter "DHCP Client" updated. • IP_DHCPC_ConfigUniBcStartMode() added. Chapter "DHCP server (Add-on)" updated. • IP_DHCPS_SetReservedAddresses() added. Chapter "Internet Protocol version 6 (IPv6) (Add-on)" updated. • IP_ICMPV6_NDP_SetDNSSLCallback() added. • IP_IPV6_SetGateway() added. Chapter "mDNS Server (Add-on)" updated. • Added "Flags" member to IP_DNS_SERVER_SD_CONFIG structure. Chapter "MQTT client (Add-on)" updated. • Added information for publisher and subscriber using the same connection. • IP_MQTT_CLIENT_Exec() added. • IP_MQTT_CLIENT_IsClientConnected() added. • IP_MQTT_CLIENT_ParsePublish() added. Chapter "WiFi support" updated. • IP_WIFI_AddClientNotificationHook() added. Chapter "Web server (Add-on)" updated. • IP_WEBS_SetErrorPageCallback() added. • IP_WEBS_AddProgressHook() added. • IP_WEBS_HEADER_AddFieldHook() added. • IP_WEBS_HEADER_CopyData() added. • IP_WEBS_HEADER_GetFindToken() added. • IP_WEBS_HEADER_SetCustomFields() added. Chapter "WebSocket (Add-on)" updated. • IP_WEBSOCKET_InitClient() added.
3.22a	0	170801	OO	Chapter "Core functions" updated. • IP_NI_PauseRxInt() added. Chapter "Discovery Add-on" updated. • Added information to structures IP_DNS_SERVER_A and IP_DNS_SERVER_AAAA when using IP address 0.
3.22	0	170728	OO	Chapter "Core functions" updated. • IP_DNS_ResolveHostEx() added. • IP_DNS_SendDynUpdate() added. • IP_DNS_SetTSIGContext() added. • IP_MDNS_ResolveHost() added. • IP_NI_AddPTPDriver() added. • IP_SendPingCheckReply() added. • IP_SetMicrosecondsCallback() added.

Software	Revision	Date	By	Description
				• IP_SetRandCallback() added. Chapter "RAW Zero-Copy" updated. • Corrected IP_RAW_SendAndFree() packet free behavior. Chapter "DHCP client" updated. • IP_DHCPC_AddStateChangeHook() added. Chapter "FTP client (Add-on)" updated. • SSL/TLS security information added. • IP_FTPC_InitEx() added. Chapter "FTP server (Add-on)" updated. • SSL/TLS security information added. • IP_FTPS_AllowOnlySecured() added. • IP_FTPS_IsDataSecured() added. • IP_FTPS_SetImplicitMode() added. Chapter "SMTP client (Add-on)" updated. • UTF-8 information added. Chapter "Socket interface" updated. • IP_SOCK_recvfrom_ts() added. • Corrected shutdown() parameters. Chapter "NTP client (Add-on)" added. Chapter "PTP Ordinary Clock (Add-on)" added. Chapter "VLAN" updated. • Added some more information to IP_VLAN_AddInterface() about VLAN Id bits. • Corrected shutdown() parameters.
3.20	0	170622	OO	Chapter "Core functions" updated. • IP_FRAGMENT_ConfigRx() added. • IP_FRAGMENT_Enable() added. • IP_IPV6_FRAGMENT_ConfigRx() added. • IP_IPV6_FRAGMENT_Enable() added. Chapter "Configuring emNet" updated. • Define IP_SUPPORT_PROFILE_FIFO added. • Define IP_SUPPORT_PROFILE_PACKET added. Chapter "Socket interface" updated. • IP_RAW_AddPacketToSocket() added. • IP_TCP_Accept() added. Chapter "mDNS Server (Add-on)" added. Chapter "DNS Server (Add-on)" added. Chapter "CoAP client/server (Add-on)" added. Chapter "Web server (Add-on)" updated. • Minor changes. Chapter "WiFi drivers" updated. • IP_NI_WIFI_REDPINE_RS9113_ConfigAntenna() added. • IP_NI_WIFI_REDPINE_RS9113_ConfigRegion() added. Chapter "Profiling with SystemView" updated.
3.16	1	170410	OO	Images in document are missing. Fixed.
3.16	0	170323	OO	Chapter "Core functions" updated. • IP_SetGlobalMcTTL() added. • IP_SetLocalMcTTL() added. Chapter "DHCP client" updated. • IP_DHCPC_ConfigDNSManually() added. Chapter "FTP server (Add-on)" updated. • IP_FTPS_ConfigBufSizes() added. • IP_FTPS_CountRequiredMem() added. • IP_FTPS_Init() added. • IP_FTPS_ProcessEx() added. • IP_FTPS_SetSignOnMsg() added. Chapter "Web server (Add-on)" updated. • IP_WEBS_ConfigFindGZipFiles() added.
3.14	0	161223	OO	Chapter "MQTT client (Add-on)" added. Chapter "WebSocket (Add-on)" added. Chapter "Socket interface" updated. • Typo in example of accept() corrected. • Typo in example of getpeername() corrected. Chapter "Web server (Add-on)" updated. • NumBytesFullUriBuf added to IP_WEBS_ConfigBufSizes() WEBS_BUFFER_SIZES sample updated. IP_WEBS_Process[Last][Ex](). • IP_WEBS_WEBSOCKET_AddHook() added. • Structure IP_WEBS_WEBSOCKET_API added. Chapter "SMTP client (Add-on)" updated.

Software	Revision	Date	By	Description
				• Updated information for sending mails with attachments (multipart messages). • Corrected ROM/RAM usage.
3.12	0	161010	OO	Chapter “Core functions” updated. • IP_NI_ClrBPressure() added. • IP_NI_SetBPressure() added. • IP_PHY_ConfigGigabitSupport() added. Chapter “PHY drivers” updated. • Marvell 88E1111 Fiber driver added. Chapter “SMTP client (Add-on)” updated. • SSL/TLS support added (IP_SMTPC_MTA/IP_SMTPC_API). Chapter “Web server (Add-on)” updated. • Digest authentication support added. • IP_WEBS_GetProtectedPath() added. • IP_WEBS_UseAuthDigest() added. • IP_WEBS_AUTH_DIGEST_CalcHA1() added. • IP_WEBS_AUTH_DIGEST_GetURI() added.
3.10	0	160912	OO	Chapter “Core functions” updated. • IP_AddLinkChangeHook() added. • IP_IsAllZero() added. • IP_NI_ConfigLinkCheckMultiplier() added. • IP_SetOnPacketFreeCallback() added. Chapter “FTP client (Add-on)” updated. • IP_FTPC_ExecCmd() added FTPC_CMD_PROT and FTPC_CMD_PBSZ added. Chapter “Web server (Add-on)” updated. • IP_WEBS_AddPreContentOutputHook() updated. • IP_WEBS_SendFormattedString() added. Chapter “WiFi support” added. Chapter “WiFi drivers” added.
3.08a	0	160712	OO	Chapter “Core functions” updated. • IP_GetCurrentLinkSpeed() updated. • IP_GetCurrentLinkSpeedEx() updated. Chapter “Web server (Add-on)” updated. • IP_WEBS_CountRequiredMem() added. • IP_WEBS_SetUploadFileSystemAPI() added. • IP_WEBS_SetUploadMaxFileSize() added. • Updated structure IP_WEBS_FILE_INFO.
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.

Software	Revision	Date	By	Description
				• IP_NI_SetAdminState() added.
3.04	0	160316	OO	Chapter “Configuring emNet” 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.
3.02	0	151125	OO	Chapter “Introduction to emNet” 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 emNet” updated. • Updated guidelines for task priorities. Chapter “Running emNet 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_DHCPC_ConfigAlwaysStartInit() added.

Software	Revision	Date	By	Description
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.
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_ConfigSelectMultiplicator() 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.

Software	Revision	Date	By	Description
				• 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.
2.12c	0	130515	OO	Chapter "Introduction to emNet" 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.

Software	Revision	Date	By	Description
				• 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 (ACD)" added.
2.02	0	120507	OO	Documentation updated for emNet 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 emNet" 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)":

Software	Revision	Date	By	Description
				• Section "FTP server system time" added. • pfGetTimeDate() added.
1.40	0	080731	SK	Chapter "API functions": • IP_TCP_SetConnKeepaliveOpt() added. • IP_TCP_SetRetransDelayRange() added. • IP_SendPacket() added. Chapter "Socket interface": • getsockopt() updated. • setsockopt() updated. Chapter "OS integration": • IP_OS_WaitItemTimed() added.
1.30	1	080610	SK	Chapter "FTP server (Add-on)" section "Resource usage" added Chapter "Web server (Add-on)" section "Resource usage" added
1.30	0	080423	SK	Chapter "FTP server (Add-on)" added. Chapter "Web server (Add-on)" updated.
1.24	3	080320	SK	Chapter "Socket interface": • getpeername added. • getsockname added.
1.24	2	080222	SK	Chapter "Device Driver": • NXP LPC23xx/24xx driver added.
1.24	1	080124	SK	Chapter "HTTP server (Add-on)" updated. Chapter "API functions": • IP_UTIL_EncodeBase64() added. • IP_UTIL_DecodeBase64() added.
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 emNet on target hardware" updated.
1.22	2	071123	SK	Chapter "Socket interface": • gethostbyname() added. • Structure hostent added. Chapter "Core functions": • IP_PrintIPAddr() added. • IP_DNS_SetServer() added.
1.22	1	071122	SK	Chapter "DHCP": • IP_DHCPC_Activate() updated. Chapter "Debugging": • Section "Testing stability" added. Chapter "Socket interface": • Section "Error codes" added.
1.22	0	071114	SK	Chapter "Introduction": • "Request for comments" enhanced. Chapter "API functions": • IP_AddLogFilter() added.

Software	Revision	Date	By	Description
				• IP_AddWarnFilter() added. • IP_GetCurrentLinkSpeed() added. • IP_TCP_Set2MSLDelay() added. • select() added. • Various function descriptions enhanced. Chapter “API functions” renamed to “core functions”. Socket functions removed from chapter “API functions” Chapter “Socket interface” added. Chapter “DHCP” added. Chapter “UDP zero copy” added. Chapter “TCP zero copy” added. Chapter “Glossary” added. Chapter “Index” updated.
1.00	2	071017	SK	Chapter “Introduction”: • Section “Features” enhanced. • Section “Basic concepts” added. • Section “Task and interrupt usage” added. • Section “Further readings” added. Chapter “Running emNet” enhanced. Chapter “API functions”: • IP_Init() added. • IP_Task() added. • IP_RxTask() added. • IP_GetVersion() added. • IP_SetLogFilter() added. • IP_SetWarnFilter() added. • IP_Panic() removed. • Structure sockaddr added. • Structure sockaddr_in added. • Structure in_addr added. Chapter “Device driver”. • General information updated. • Section “Writing your own driver” added. Chapter “Debugging” added. Chapter “Performance and resource usage” added. Chapter “OS integration” updated.
1.00	1	071002	SK	Product name changed to “emNet”: Chapter “API functions”: • IP_X_Prepare() renamed to IP_X_Config(). • IP_AddBuffers() added. • IP_ConfTCPSpace() added.
1.00	0	070927	SK	Initial version.

About this document

Assumptions

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

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

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Ritchie (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 program 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 of contents

Chapter 1

Introduction to emNet

This chapter provides an introduction to using emNet. It explains the basic concepts behind emNet.

1.1 What is emNet

emNet is a CPU-independent TCP/IP stack.

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

1.2 Features

emNet is written in ANSI C and can be used on virtually any CPU.

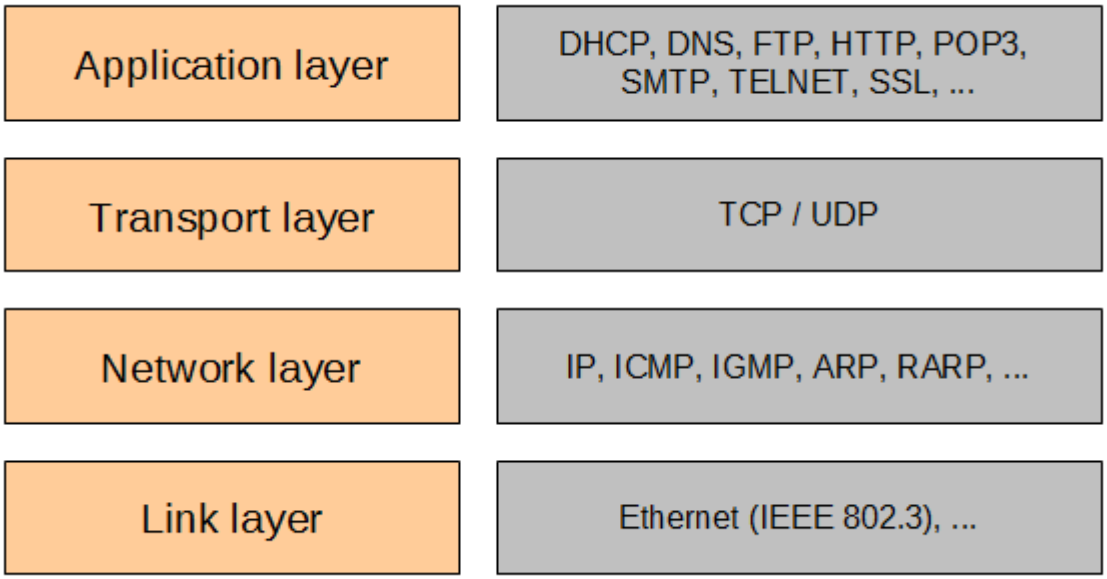
Some features of emNet:

- 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 emNet structure

emNet 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 emNet and the user application. It uses the emNet API to transmit data over an TCP/IP network. The emNet 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.

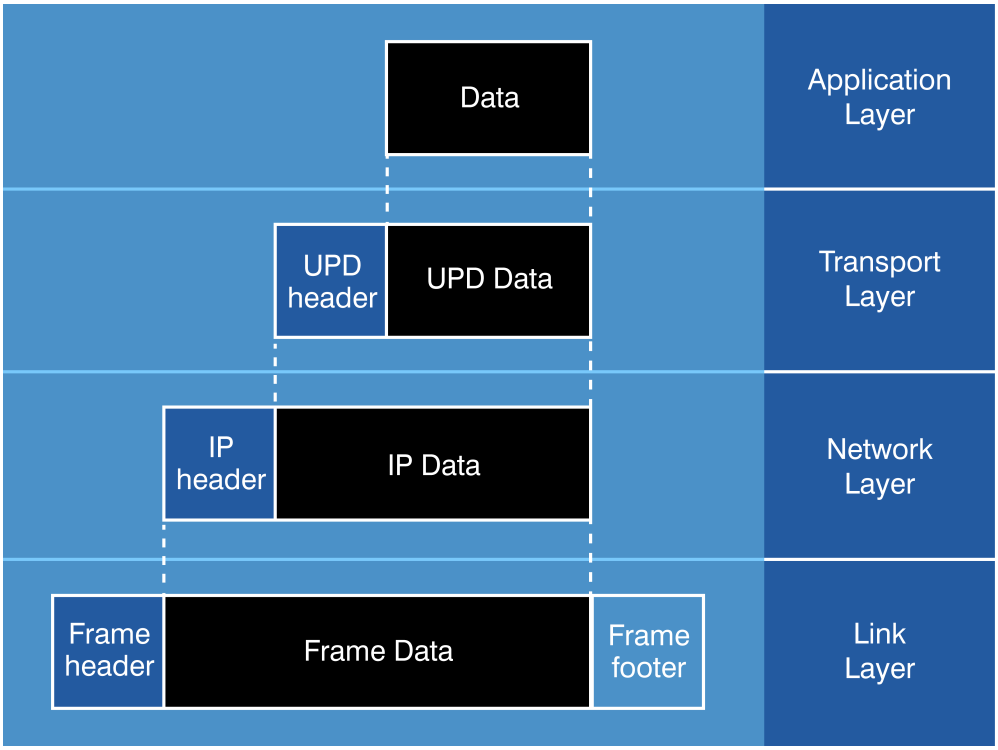
Link layer

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 emNet 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 a 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

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

Task priorities

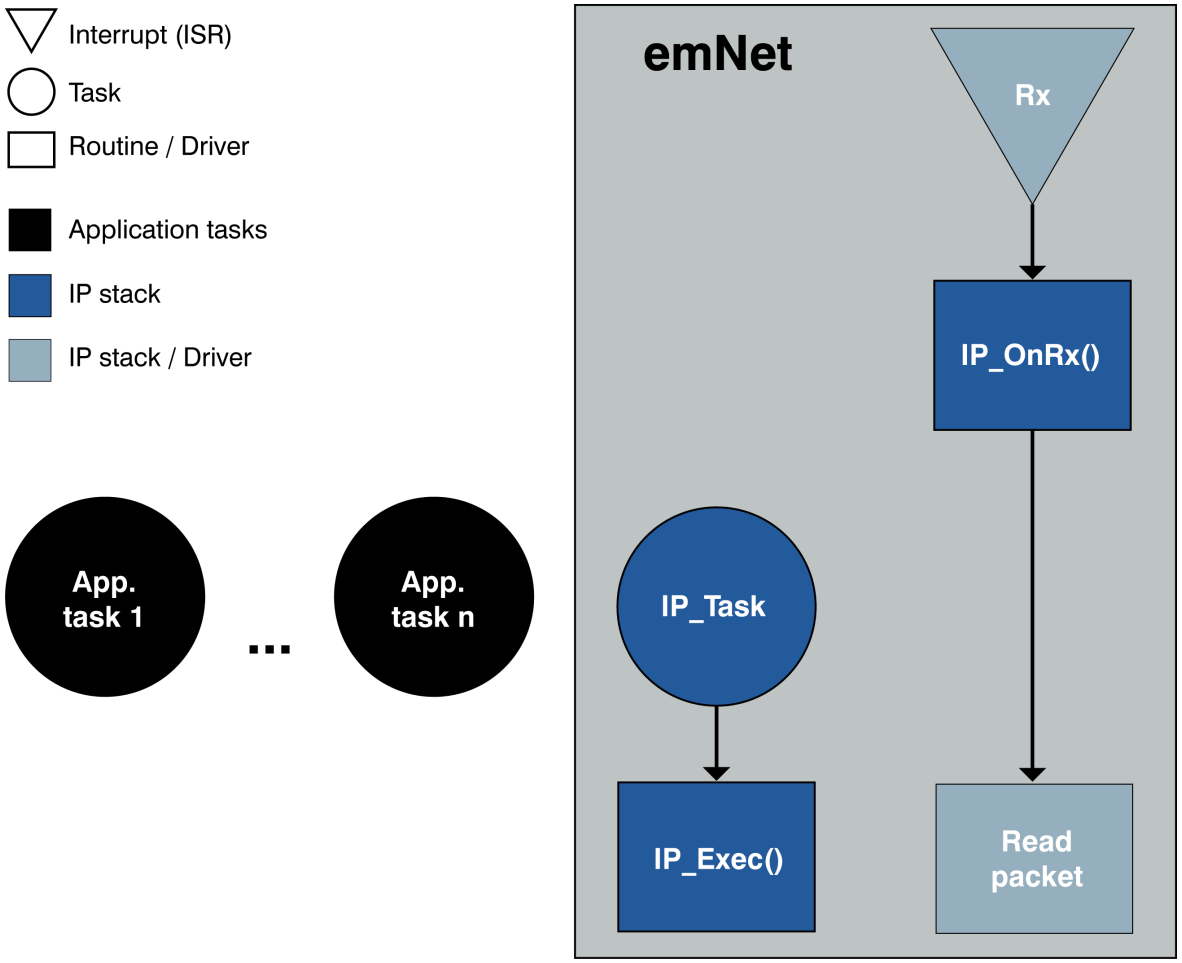
IP task priorities are independent from other (non IP) task priorities. However as soon as a task calls an IP API it should follow these priority rules for the best performance of the stack:

1. The `IP_RxTask` (if used at all) should 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 should 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.

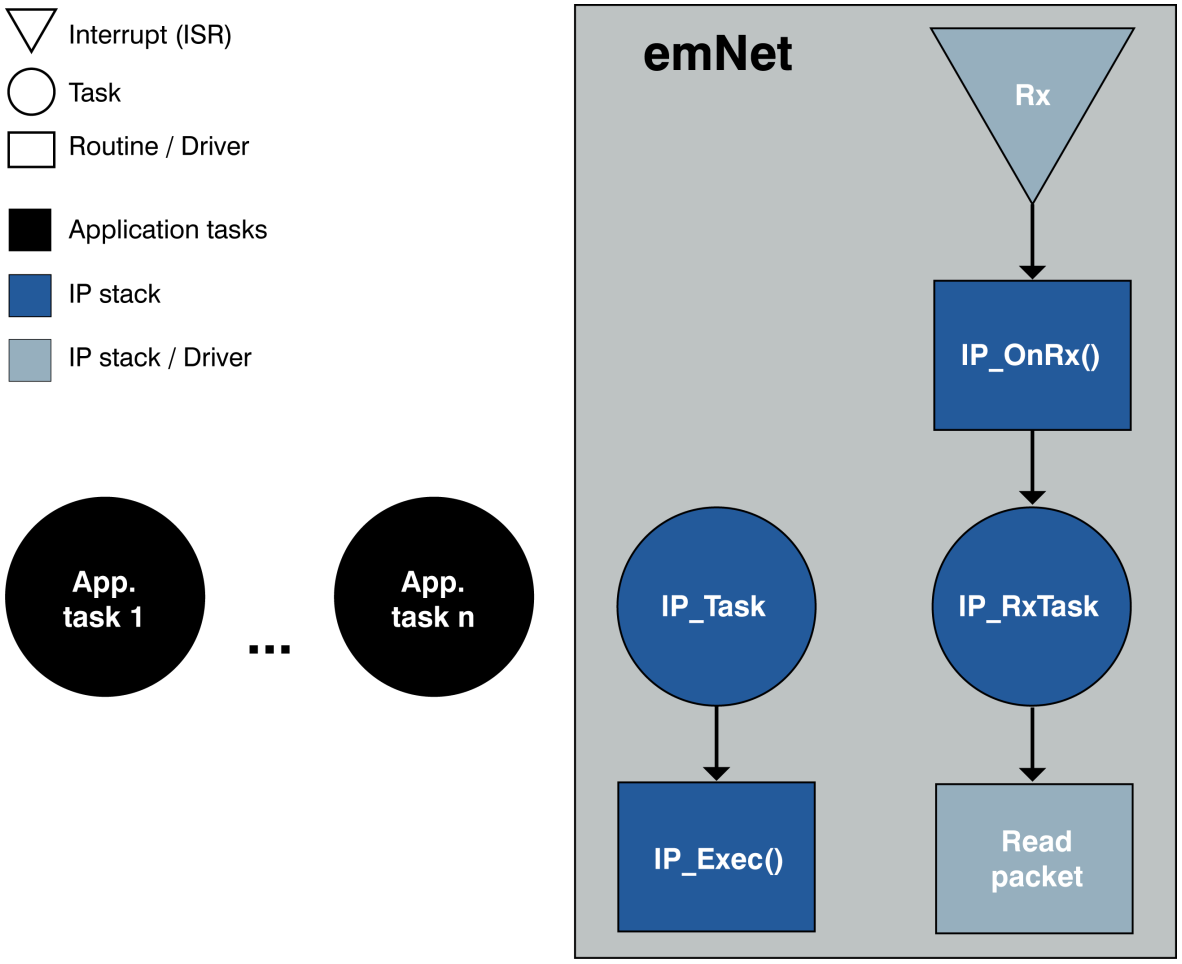
One task dedicated to the stack

Using 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 166 for more information and an example about how to include the `IP_Task` into your project.



Two tasks 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 woken 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 166 and *IP_RxTask* on page 173 for more information. IP_RxTask should have a higher priority than IP_Task as it is treated as interrupt in task form and should not be interrupted by IP_Task or any other IP task.



Note

Initializing the IP stack with two task concept from main()

Packets might receive as soon as Ethernet is initialized by IP_Init() and the (receive) interrupt is enabled. The internal switch between the single task and two task concept gets set automatically upon the first execution of IP_RxTask() . Initializing the stack from main() typically means to initialize it before a task scheduler is active.

If a packet is received between IP_Init() and the first run of IP_RxTask() the packet will be processed like in the single task concept. This should typically cause no problem to the application and the mode is automatically switched to the two task concept as soon as IP_RxTask() has run for the first time.

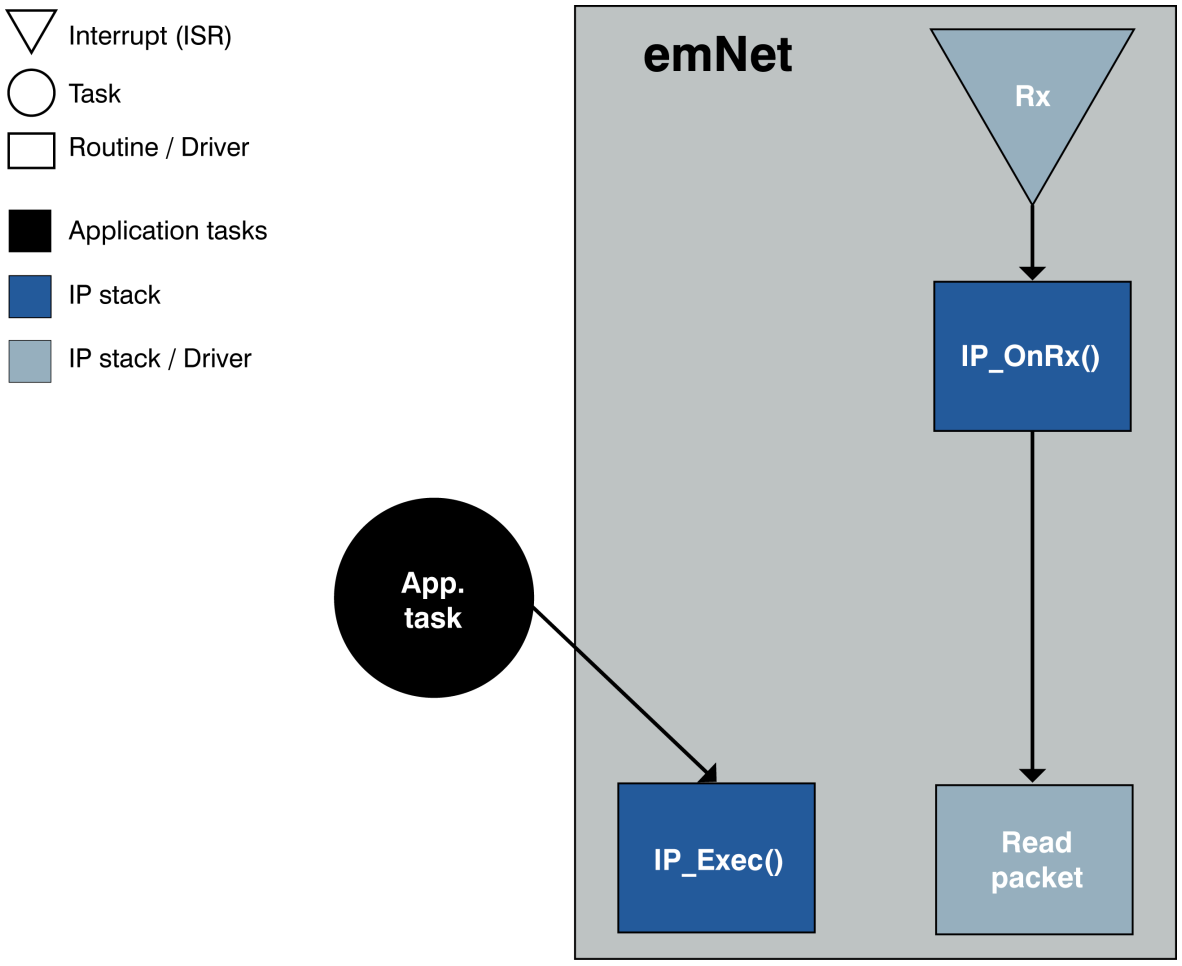
If it is desired to completely avoid this to happen, the following steps need to be taken care of:

1. Call IP_SetUseRxTask() manually (best placed after calling IP_Init() and adding the two tasks) to switch to the two task concept early.

2. The Ethernet interrupt enable (typically in `BSP_IP.c`) needs to be manually moved to after calling `IP_Init()` and `IP_SetUseRxTask()` .

Zero tasks dedicated to the stack (Superloop)

emNet 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.



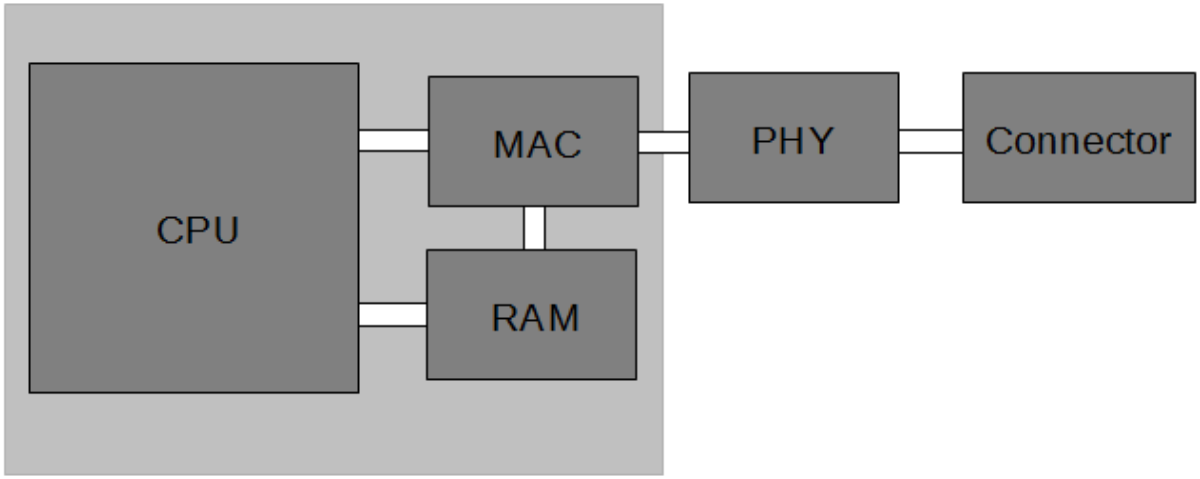
1.5 Background information

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.

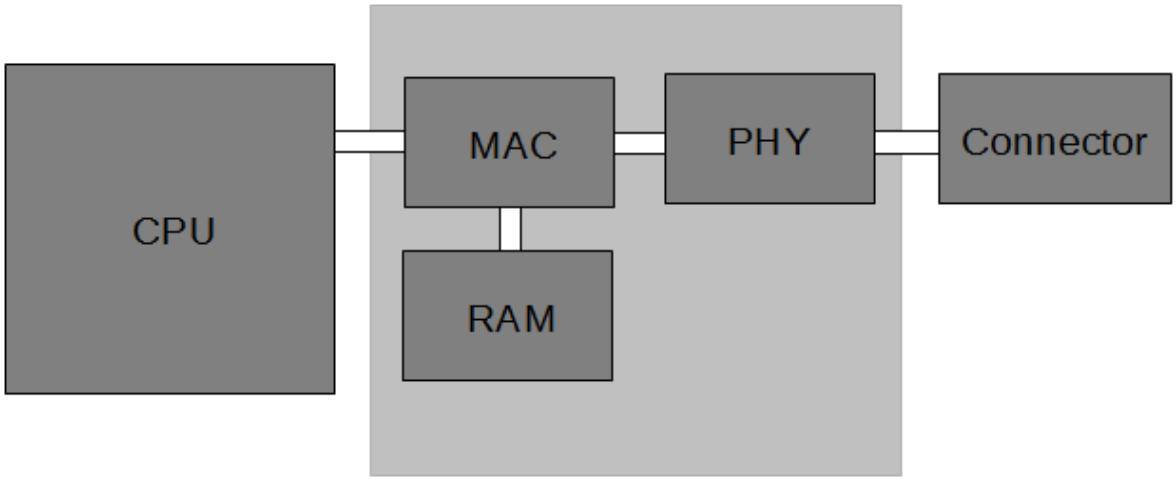
MCUs with integrated MAC

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.

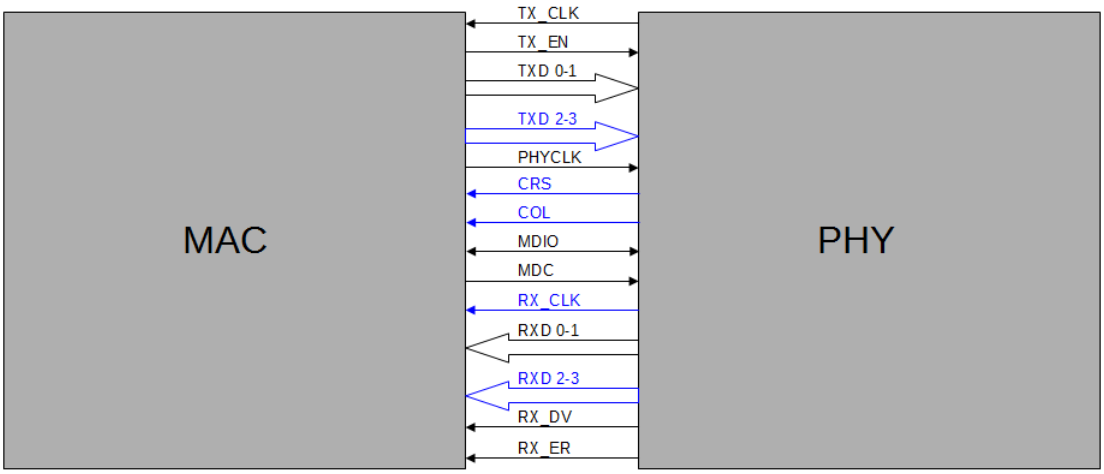
In theory, up to 32 PHYs can be connected to a single MAC. In praxis, this is never done; only one PHY is connected. In order to allow multiple PHYs to be connected to a single MAC, individual 5-bit addresses have to be assigned to the different PHYs. If only one PHY is connected, the emNet driver automatically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

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]`	4-bit Transmit Data (cont'd)	N/A
`PHYCLK`	PHY Clock Output	PHY Clock Output
`CRS`	Carrier Sense	N/A

Signal	MII	RMII
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]	4-bit Receive Data (cont’d)	N/A
RX_DV	Data Valid	Carrier Sense/Data Valid
RX_ER	Receive Error	Receive Error

1.6 Further reading

This guide explains the usage of the emNet 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]	B00TP - 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 emNet on target hardware

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

Integrating emNet

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

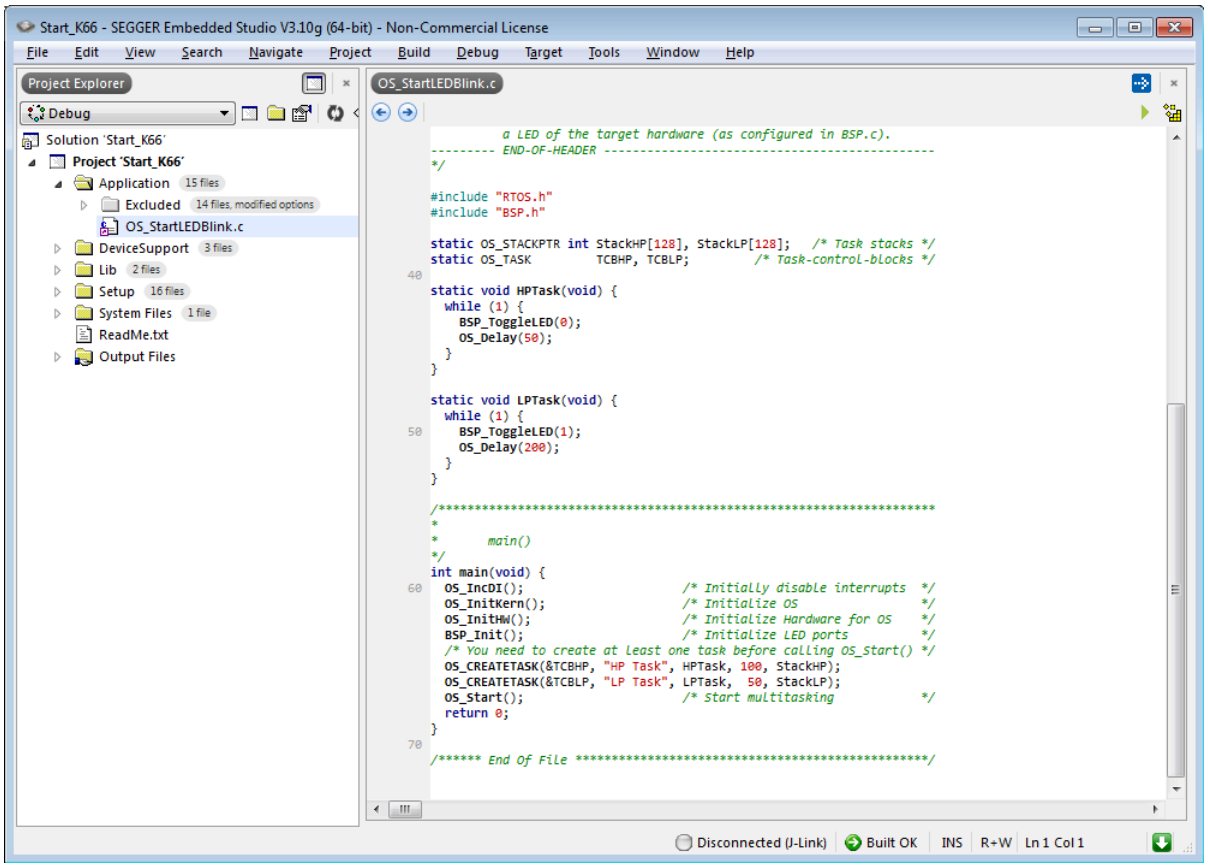
Procedure to follow

Integration of emNet is a relatively simple process, which consists of the following steps:

- Step 1: Open an embOS project and compile it.
- Step 2: Add emNet 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 emNet to the start project

Add all IP source files as in the IP release delivery to your project.

The `Config` folder includes all configuration files of emNet. The configuration files are pre-configured 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` 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 `SEGGER` folder is an optional component of the emNet shipment. It contains optimized MCU and/or compiler specific files, for example a special memcopy function.

BSP support

IP drivers need hardware setting from the BSP file (like port settings for example). Some older driver are supplied with `BSP.c` and `BSP.h` that need to replace the one supplied with embOS shipment.

Newer and updated drivers have a separate `BSP_IP.c` file instead.

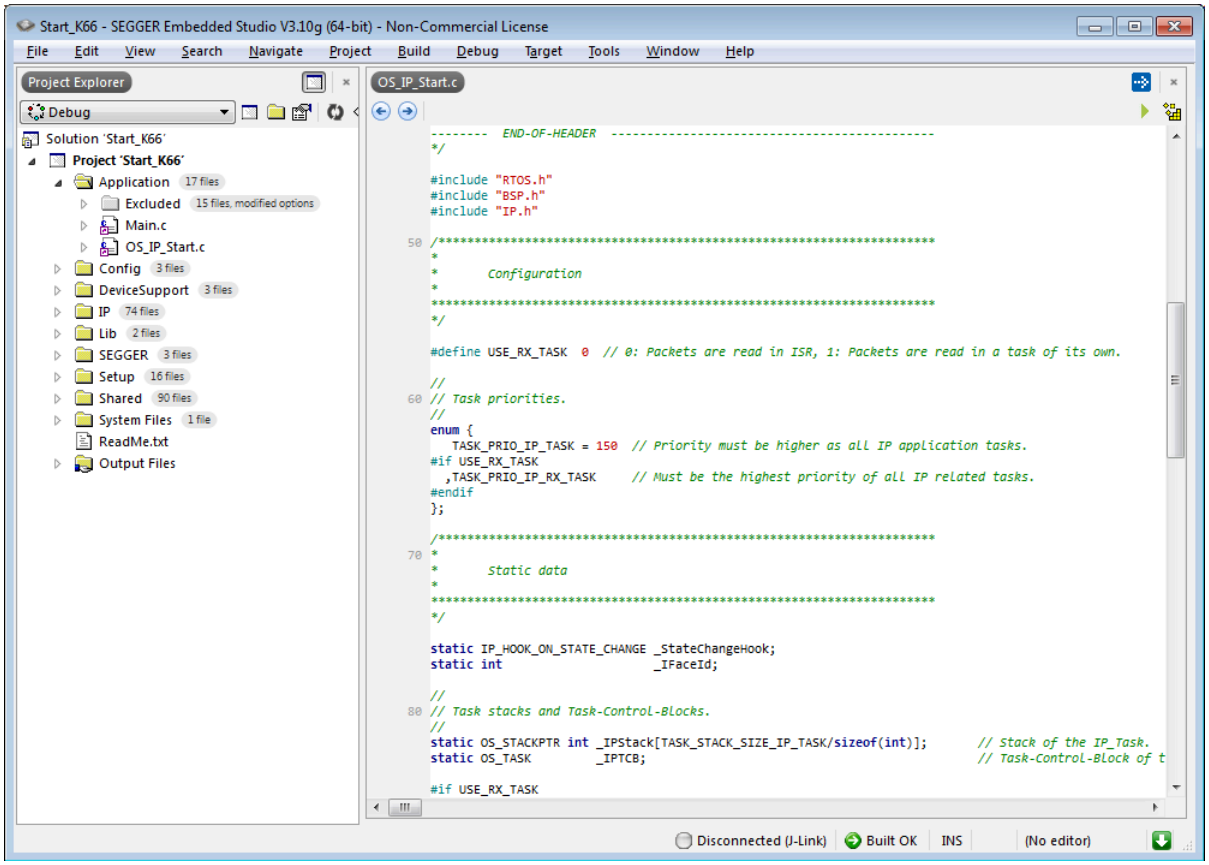
Depending on your case, either replace `BSP.c` and `BSP.h` of your embOS start project or add `BSP_IP.c`.

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 IP directories to your include path.

Select the start application

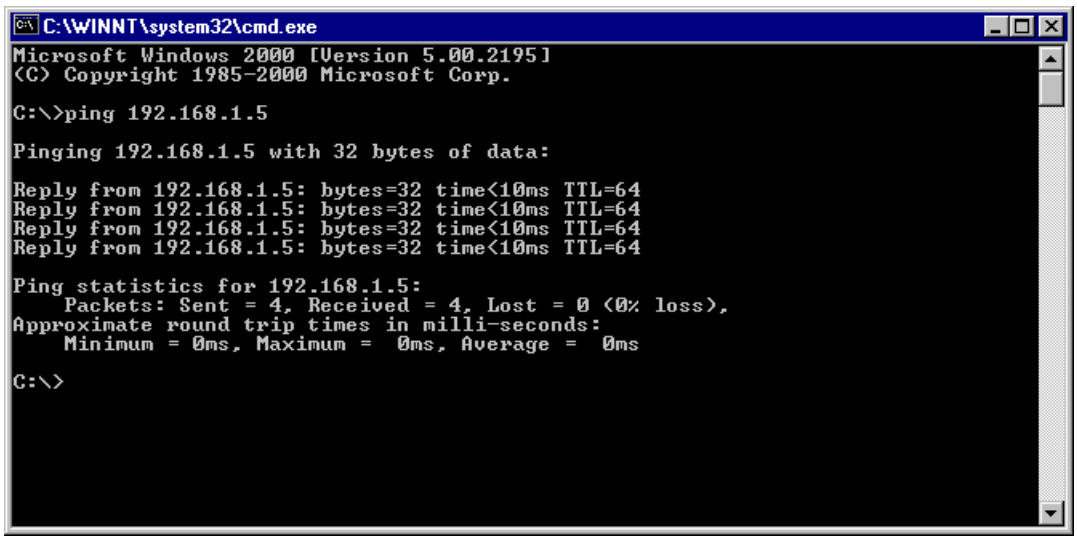
For quick and easy testing of your emNet integration, start with the code found in the folder `Application`. Add one of the applications to your project (for example `IP_SimpleServer.c`).



2.3 Step 3: Build the project and test it

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

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



Chapter 3

Example applications

In this chapter, you will find a description of each emNet example application.

3.1 Overview

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

The following start application files are provided:

File	Description
`IP_DNSClient.c`	Demonstrates the use of the integrated DNS client.
`IP_NonBlockingConnect.c`	Demonstrates how to connect to a server using non-blocking sockets.
`IP_Ping.c`	Demonstrates how to send ICMP echo requests and how to process ICMP replies in application.
`IP_SHELL_Start.c`	Demonstrates using the IP-shell to diagnose the IP stack.
`IP_SimpleServer.c`	Demonstrates setup of a simple server which simply sends back the target system tick for every character received.
`IP_SpeedClient_TCP.c`	Demonstrates the TCP send and receive performance of the device running emNet.
`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 `192.168.2.252` if the DHCP client is not enabled.
`IP_UDPDiscover.c`	Demonstrates setup of a simple UDP application which replies to UDP broadcasts. The application sends an answer for every received discover packet. The related host application sends discover packets as UDP broadcasts and waits for the feedback of the targets which are available in the subnet.
`IP_UDPDiscover_ZeroCopy.c`	Demonstrates setup of a simple UDP application which replies to UDP broadcasts. The application uses the the emNet 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.

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

3.1.1 emNet DNS client (IP_DNSClient.c)

The emNet 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 emNet non-blocking connect (IP_NonBlockingConnect.c)

The emNet non-blocking connect sample demonstrates how to connect to a server using non-blocking sockets. The target tries to connect to TCP server with an non-blocking socket. The sample can be used with any TCP server independent of the application which is listening on the port. The client only opens a TCP connection to the server and closes it

without any further communication. The terminal I/O output in your debugger should be similar to the following out:

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

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

3.1.3 emNet ping (IP_Ping.c)

The emNet 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 emNet ICMP implementation, you have to perform the following steps:

1. Customize the `Local defines, configurable` section of `IP_Ping.c`. Change the macro `HOST_TO_PING` according 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:

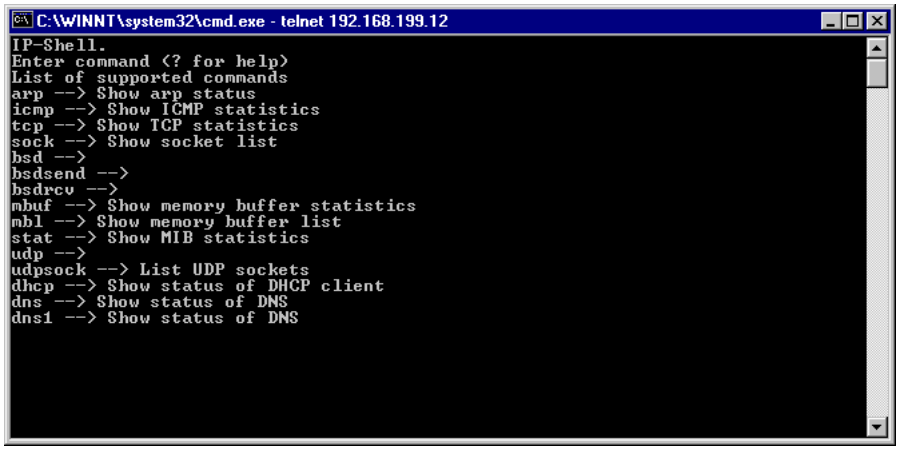
```
ping [IP_ADDRESS _OF_YOUR_TARGET_RUNNING_EMNET]
```

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 emNet shell (IP_SHELL_Start.c)

The emNet 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 `192.168.2.252` if the DHCP client is not enabled.



3.1.5 emNet simple server (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 `192.168.2.252` if the DHCP client is not enabled.

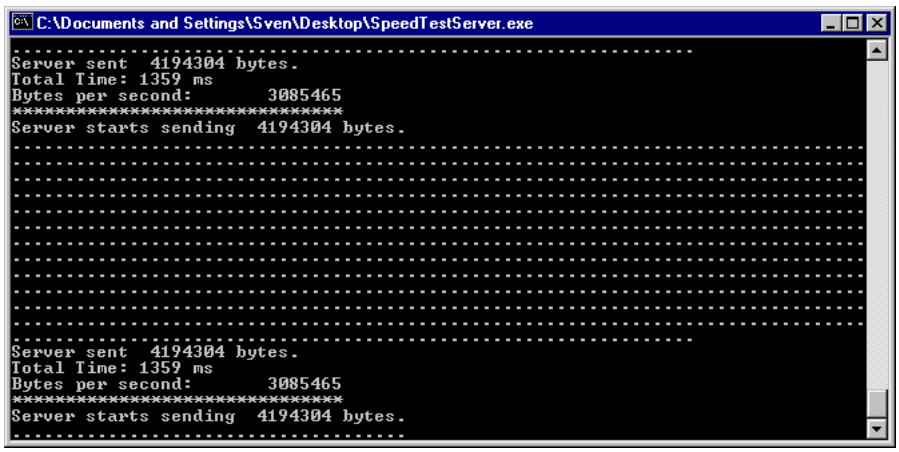
3.1.6 emNet speed client (IP_SpeedClient_TCP.c)

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

3.1.6.1 Running the emNet speed client

To test the emNet 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 `IP_SpeedClient.c` to your project.
3. Customize the `Local defines, configurable` section of `IP_SpeedClient.c`. Change the macro `SERVER_IP_ADDR` according to your configuration. For example, if the Windows host PC running the speed test server uses the IP address 192.168.5.15, change the `SERVER_IP_ADDR` macro to `0xC0A8050F`. If you have changed the port which the Windows host application uses to listen, change the macro `SERVER_PORT` accordingly.
4. Build and download the speed client into your target. The target connects to the server and starts the transmission.



3.1.7 emNet start (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 `192.168.2.252` if the DHCP client is not enabled.

3.1.8 emNet UDP discover (IP_UDPDiscover.c / IP_UDPDiscover_ZeroCopy.c)

To test the emNet 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 `IP_UDPDiscover.c` to your project.
3. Customize the `Local defines, configurable` section of `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 emNet core function.

4.1 API functions

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

Function	Description
Configuration functions	
IP_AddBuffers()	Adds buffers to the TCP/IP stack.
IP_AddEtherInterface()	Adds an Ethernet interface to the stack.
IP_AddVirtEtherInterface()	Adds a virtual interface to the stack that uses a hardware interface for communication.
IP_AddLoopbackInterface()	Adds a loopback interface to the stack.
IP_AddMemory()	This function is called from the application to add additional memory to the stack.
IP_AllowBackPressure()	Allows back pressure if the driver supports this feature.
IP_AssignMemory()	Assigns memory to the stack.
IP_ARP_ConfigAgeout()	Configures the timeout for cached ARP entries.
IP_ARP_ConfigAgeoutNoReply()	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.
IP_ARP_ConfigAgeoutSniff()	Configures the age out value for ARP entries, which we have created by looking up addresses of received IP packets.
IP_ARP_ConfigAllowGratuitousARP()	Configures if gratuitous ARP packets from other network members are allowed to update the ARP cache.
IP_ARP_ConfigAnnounceStaticIP()	Configures whether to announce using a static IP in the network using gratuitous ARP packets.
IP_ARP_ConfigMaxPending()	Configures the maximum number packets that can be queued waiting for an ARP reply.
IP_ARP_ConfigMaxRetries()	Configures how often an ARP request is resent before considering the request failed.
IP_ARP_ConfigNumEntries()	Configures the maximum number of possible entries in the ARP cache.
IP_BSP_SetAPI()	Sets an API to be used for BSP related abstraction like initializing hardware and installing interrupt handlers.
IP_ConfigDoNotAddLowLevelChecks_ARP()	Tells the stack to not add low level ARP checks when initializing the stack with IP_Init().
IP_ConfigDoNotAddLowLevelChecks_UDP()	Tells the stack to not add low level UDP checks when initializing the stack with IP_Init().
IP_ConfigMaxIFaces()	Configures the maximum number of interfaces that can be added to the system.
IP_ConfigNumLinkDownProbes()	Configures the number of continuous link down probes to take before the stack accepts the link down status.

Function	Description
IP_ConfigNumLinkUpProbes()	Configures the number of continuous link up probes to take before the stack accepts the link up status.
IP_ConfigOffCached2Uncached()	Configures the offset from a cached memory area to its uncached equivalent for uncached access.
IP_ConfigReportSameMacOnNet()	Configures if the stack warns about receiving an Ethernet packet from the same HW address as the interface the packet came in.
IP_ConfigTCPSpace()	Configures the size of the TCP send and receive window size.
IP_DisableIPRxChecksum()	Disables checksum verification of the checksum in the IP header for incoming packets.
IP_DisableIPv4()	Disables IPv4 in the stack as good as possible.
IP_CACHE_SetConfig()	Configures cache related functionality that might be required by the stack for several purposes such as cache handling in drivers.
IP_DNS_GetServer()	Retrieves the first DNS server configured of the first interface.
IP_DNS_GetServerEx()	Retrieves a DNS server configured for an interface.
IP_DNS_ResolveHostEx()	Sends a query to the DNS server.
IP_DNS_SendDynUpdate()	Build a dynamic update request.
IP_DNS_SetTSIGContext()	Set the TSIG signature context with the parameters needed to perform Secured Dynamic Updates signed with TSIG.
IP_DNS_SetMaxTTL()	Sets the maximum Time To Live (TTL) of a DNS entry in seconds.
IP_DNS_SetServer()	Sets the DNS server address of the first interface.
IP_DNS_SetServerEx()	Sets the IP address of the available DNS servers for an interface.
IP_MDNS_ResolveHost()	Sends a query using Multicast DNS.
IP_MDNS_ResolveHostSingleIP()	Sends a query using Multicast DNS.
IP_EnableIPRxChecksum()	Enables the IP Rx checksum calculation in the IP header for incoming packets.
IP_GetMaxAvailPacketSize()	Asks the stack for the maximum available free packet size that can then be allocated.
IP_GetMemPoolInfo()	Collects data about a memory pool such as its size and free bytes.
IP_GetMTU()	Retrieves the configured TCP MTU size for an interface.
IP_GetPrimaryIFace()	Retrieves the currently set primary interface index.
IP_ICMP_Add()	Adds ICMP Protocol function to the stack.
IP_ICMP_DisableRxChecksum()	Disables the ICMP Rx checksum calculation.

Function	Description
IP_ICMP_EnableRxChecksum()	Enables the ICMP Rx checksum calculation.
IP_IGMP_Add()	Adds the IGMP protocol to interface #0.
IP_IGMP_AddEx()	Adds the IGMP protocol to an interface.
IP_IGMP_ConfigV2AlwaysReport()	Configures if upon every IGMPv2 QUERY a REPORT shall be sent back.
IP_IGMP_JoinGroup()	Joins an IGMP group.
IP_IGMP_JoinGroup_AutoRejoin()	Joins an IGMP group and rejoins when the interface link state changes.
IP_IGMP_LeaveGroup()	Leaves an IGMP group.
IP_RAW_Add()	Adds RAW socket function to stack.
IP_SetAddrMask()	Sets the IP address and subnet mask of an interface.
IP_SetAddrMaskEx()	Sets the IP address and subnet mask of an interface.
IP_SetGWAddr()	Sets the default gateway address of the selected interface.
IP_SetHWAddr()	Sets the Media Access Control address (MAC) of the interface 0.
IP_SetHWAddrEx()	Sets the Media Access Control address (MAC) of the selected interface.
IP_SetMTU()	Allows to set the maximum transmission unit (MTU) of an interface.
IP_SetMicrosecondsCallback()	Sets a callback that is used to get a time-stamp in microseconds.
IP_SetNanosecondsCallback()	Sets a callback that is used to get a time-stamp in nanoseconds.
IP_SetOnIFaceSelectCallback()	Sets a callback that gets notified about an internal interface selection by the stack and allows to override it.
IP_SetPrimaryIFace()	Sets the primary interface index.
IP_SetSupportedDuplexModes()	Sets the supported duplex/speed of the device to be advertised during Auto-Negotiation.
IP_SetTTL()	Sets the default value for the Time-To-Live IP header field.
IP_SetGlobalMcTTL()	Sets the default value for the Time-To-Live IP header field for global multicast packets.
IP_SetLocalMcTTL()	Sets the default value for the Time-To-Live IP header field for local multicast packets.
IP_SetUseRxTask()	Sets the internal flag for using the IP_RxTask() manually.
IP_SOCKET_ConfigSelectMultiplicator()	Configures the multiplicator for the time-out parameter of select().
IP_SOCKET_SetDefaultOptions()	Sets the socket options enabled by default.
IP_SOCKET_SetLimit()	Sets the maximum number of allowed sockets.

Function	Description
IP_SYSVIEW_Init()	Initializes the profile instrumentation of the stack and SystemView as profiling implementation.
IP_TCP_Add()	Adds TCP Protocol function to the stack.
IP_TCP_DisableRxChecksum()	Disables the TCP Rx checksum calculation.
IP_TCP_EnableRxChecksum()	Enables checksum verification of the checksum in the TCP header for incoming packets.
IP_TCP_Set2MSLDelay()	Sets the maximum segment lifetime (MSL).
IP_TCP_SetConnKeepaliveOpt()	Sets the keepalive options.
IP_TCP_SetRetransDelayRange()	Sets retransmission delay range.
IP_UDP_Add()	Adds UDP Protocol support to the stack.
IP_UDP_AddEchoServer()	Adds a simple echo server for UDP packets that can be used for UDP pings and other tests.
IP_UDP_DisableRxChecksum()	Disables checksum verification of the checksum in the UDP header for incoming packets.
IP_UDP_EnableRxChecksum()	Enables checksum verification of the checksum in the TCP header for incoming packets.
Configuration functions (IP fragmentation)	
IP_FRAGMENT_ConfigRx()	Modifies the default settings for IPv4 fragmentation.
IP_FRAGMENT_Enable()	Initializes the required variables and adds a timer to the stack to handle outdated fragment queues.
IP_IPV6_FRAGMENT_ConfigRx()	Modifies the default settings for IPv6 fragmentation.
IP_IPV6_FRAGMENT_Enable()	Initializes the required variables and adds a timer to the stack to handle outdated fragment queues.
Management functions	
IP_DeInit()	Deinitializes the TCP/IP stack.
IP_Init()	Initializes the TCP/IP stack.
IP_Task()	Main task for handling the stack.
IP_Exec()	Processes received packets and handles timers and other jobs.
IP_TASK_Init()	Initializes the main IP task context when not using IP_Task().
IP_TASK_Exec()	Processes received packets and handles timers and other jobs.
IP_TASK_WaitForEvent()	Waits for an event for the main IP task to be signaled.
IP_RxTask()	Optional task to reduce time spent in receive interrupts.
IP_RXTASK_Init()	Initializes the RxTask context when not using IP_RxTask().

Function	Description
IP_RXTASK_Exec()	Copies received packets from driver to stack in a task context instead of from an interrupt.
IP_RXTASK_WaitForEvent()	Waits for an event for the IP_RxTask to be signaled.
IP_Shutdown()	Prepare network stack related tasks for a graceful shutdown.
Network interface configuration and handling functions	
IP_NI_AddPTPDriver()	Adds an NI specific PTP driver for HW timestamp support.
IP_NI_ClrBPressure()	Disables usage of back pressure (sending a jam signal to signal when we run into a shortage where the hardware can not receive more data).
IP_NI_ConfigPoll()	Select polled mode for the network interface.
IP_NI_ForceCaps()	Allows to force capabilities to be set for an interface.
IP_NI_SetBPressure()	Enables usage of back pressure (sending a jam signal to signal when we run into a shortage where the hardware can not receive more data).
IP_NI_SetTxBufferSize()	Sets the size of the Tx buffer of the network interface.
PHY configuration functions	
IP_NI_ConfigPHYAddr()	Configure the PHY Addr.
IP_NI_ConfigPHYMode()	Configure the PHY mode.
IP_PHY_AddDriver()	Adds a PHY driver and assigns it to an interface.
IP_PHY_AddResetHook()	This function adds a hook function to the IP_HOOK_ON_PHY_RESET list.
IP_PHY_ConfigAddr()	Configures the PHY address to use.
IP_PHY_ConfigAfterResetDelay()	Configures the delay between (soft) resetting the PHY and further communication with it.
IP_PHY_ConfigAltAddr()	Sets a list of PHY addresses that can alternately be checked for the link state.
IP_PHY_ConfigGigabitSupport()	Configures if the MAC supports Gigabit Ethernet.
IP_PHY_ConfigSupportedModes()	Configures the supported duplex/speed of the device to be advertised during Auto-Negotiation.
IP_PHY_ConfigUseStaticFilters()	Tells the stack if using PHY static MAC filter is allowed.
IP_PHY_DisableCheck()	Disables PHY checks for all interfaces.
IP_PHY_DisableCheckEx()	Disables PHY checks for one interface.
IP_PHY_ReadReg()	Reads a PHY register.
IP_PHY_ReInit()	Re-initializes the PHY.
IP_PHY_SetWdTimeout()	Sets the watchdog timeout for watching if the PHY reached an unstable state.

Function	Description
IP_PHY_WriteReg()	Writes a PHY register.
Statistics functions	
IP_STATS_EnableIFaceCounters()	Enables statistic counters for a specific interface.
IP_STATS_GetIFaceCounters()	Retrieves a pointer to the statistic counters for a specific interface.
IP_STATS_GetLastLinkStateChange()	Retrieves the tick count when an interface entered its current state.
IP_STATS_GetRxBytesCnt()	Retrieves the number of bytes received on an interface.
IP_STATS_GetRxDiscardCnt()	Retrieves the number of packets received but discarded although they were O.K.
IP_STATS_GetRxErrCnt()	Retrieves the number of receive errors.
IP_STATS_GetRxNotUnicastCnt()	Retrieves the number of packets received on an interface that were not unicasts.
IP_STATS_GetRxUnicastCnt()	Retrieves the number of unicast packets received on an interface.
IP_STATS_GetRxUnknownProtoCnt()	Retrieves the number of unknown protocols received.
IP_STATS_GetTxBytesCnt()	Retrieves the number of bytes sent on an interface.
IP_STATS_GetTxDiscardCnt()	Retrieves the number of packets to send but discarded although they were O.K.
IP_STATS_GetTxErrCnt()	Retrieves the number of send errors on an interface.
IP_STATS_GetTxNotUnicastCnt()	Retrieves the number of packets sent on an interface that were not unicasts.
IP_STATS_GetTxUnicastCnt()	Retrieves the number of unicast packets sent on an interface.
Other IP stack functions	
IP_AddAfterInitHook()	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.
IP_AddEtherTypeHook()	This function registers a callback to be called for received packets with the registered Ethernet type.
IP_AddInterfaceErrorHook()	Adds a hook function which will be called if initialization fails for an interface.
IP_AddLinkChangeHook()	Adds a callback that gets executed each time the link state changes.
IP_AddOnPacketFreeHook()	This function adds a hook function to the IP_HOOK_ON_PACKET_FREE list.
IP_AddStateChangeHook()	Adds a hook to a callback that is executed when the AdminState or HWState of an interface changes.
IP_Alloc()	Thread safe memory allocation from main IP stack memory pool.

Function	Description
IP_AllocEtherPacket()	Allocates a packet to store the raw data of an Ethernet packet of up to NumBytes at the location returned by ppBuffer.
IP_AllocEx()	Thread safe memory allocation from a specific memory pool managed by the stack that has been added using IP_AddMemory().
IP_ARP_CleanCache()	Cleans all ARP entries that are not static entries.
IP_ARP_CleanCacheByInterface()	Cleans all ARP entries that are known to belong to a specific interface and are not static entries.
IP_Connect()	Calls a previously registered hook for the interface if any was set using IP_SetIFaceConnectHook().
IP_Disconnect()	Calls a previously registered hook for the interface if any was set using IP_SetIFaceDisconnectHook().
IP_Err2Str()	Converts IP stack error code to a readable string by simply using the defines name.
IP_FindIFaceByIP()	Tries to find out the interface number when only the IP address is known.
IP_Free()	Thread safe memory free to IP stack memory pools.
IP_FreePacket()	Frees a packet back to the stack.
IP_GetAddrMask()	Retrieves the IP address and subnet mask of an interface.
IP_GetCurrentLinkSpeed()	Returns the current link speed of the first interface (interface ID 0).
IP_GetCurrentLinkSpeedEx()	Returns the current link speed of the requested interface.
IP_GetFreePacketCnt()	Checks how many packets for a specific size or greater are currently available in the system.
IP_GetIFaceHeaderSize()	Retrieves the size of the header necessary for the transport medium that is used by a specific interface.
IP_GetGWAddr()	Returns the gateway address of the interface in host endianness.
IP_GetHWAddr()	Returns the hardware address (Media Access Control address) of the interface.
IP_GetIPAddr()	Returns the IP address of the interface in host endianness.
IP_GetIPPacketInfo()	Returns the start address of the data part of an IPv4 packet.
IP_GetRawPacketInfo()	Returns the start address of the raw data of an IP_PACKET.
IP_GetVersion()	Returns the version of the stack.
IP_ICMP_AddRxHook()	This function adds a callback that is executed upon receiving an ICMPv4 packet.

Function	Description
IP_ICMP_SetRxHook()	Sets a hook function which will be called if target receives a ping packet.
IP_ICMP_RemoveRxHook()	This function removes a hook function from the IP_HOOK_ON_ICMPV4 list.
IP_IFaceIsReady()	Checks if the interface is ready for usage.
IP_IFaceIsReadyEx()	Checks if the specified interface is ready for usage.
IP_IsAllZero()	Checks if there are zeros at the given pointer.
IP_IsExpired()	Checks if the given system timestamp has already expired.
IP_NI_ConfigLinkCheckMultiplier()	Configures the multiplier of the period between interface link checks typically executed each second.
IP_NI_ConfigUsePromiscuousMode()	Configures if the driver tries to use its hardware precise and hash filters as available before switching to promiscuous mode or if promiscuous mode will be used in any case.
IP_NI_GetAdminState()	Retrieves the admin state of the given interface.
IP_NI_GetIFaceType()	Retrieves a short textual description of the interface type.
IP_NI_GetState()	Returns the hardware state of the interface.
IP_NI_SetAdminState()	Sets the AdminState of the interface.
IP_NI_GetTxQueueLen()	Retrieves the current length of the Tx queue of an interface.
IP_NI_PauseRx()	Pauses the Rx handling of an interface by disabling it temporary.
IP_NI_PauseRxInt()	Pauses the Rx interrupt of an interface by disabling it temporary.
IP_PrintIPAddr()	Convert a 4-byte IP address to a dots-and-number string.
IP_ResolveHost()	Resolve a host name string to its IP address by using a configured DNS server.
IP_RemoveEtherTypeHook()	This function removes a hook function for a previously registered Ethernet type.
IP_SendEtherPacket()	Sends a previously allocated Ethernet packet.
IP_SendPacket()	Sends a user defined packet on the interface.
IP_SendPing()	Sends a single ICMP echo request (“ping”) to the specified host.
IP_SendPingCheckReply()	Sends a single ICMP echo request (“ping”) to the specified host using the selected interface and waits for the reply.
IP_SendPingEx()	Sends a single ICMP echo request (“ping”) to the specified host using the selected interface.

Function	Description
IP_SetIFaceConnectHook()	Sets a hook for an interface that is executed when IP_Connect() is called.
IP_SetIFaceDisconnectHook()	Sets a hook for an interface that is executed when IP_Disconnect() is called.
IP_SetOnPacketFreeCallback()	This function sets a callback to be executed once the packet has been freed.
IP_SetPacketToS()	Sets the value of the ToS/DSCP byte in the IP header of a packet to be sent via the zero-copy API.
IP_SetRxHook()	Sets a hook function which will be called if target receives a packet.
IP_SetRandCallback()	Sets a callback that can provide random data.

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);
```

Parameters

Parameter	Description
`NumBuffers`	The number of buffers.
`BytesPerBuffer`	Size of buffers in bytes.

Additional information

The stack 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 to the stack.

Prototype

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

Parameters

Parameter	Description
pDriver	Pointer to a network interface driver structure.

Return value

≥ 0 Zero-based interface index of the newly created interface.
< 0 Error.

Additional information

Optional configuration of the maximum number of interfaces that can be added to the system using `IP_ConfigMaxIFaces()` needs to be done before adding any interface and must not be changed later.

While the order in which interfaces are added to the stack does not matter to the stack itself, it might be important for the driver to add.

Typically drivers for CPU integrated controllers are expected to be added first. Next drivers for external controllers can be added. As external controllers can be used as an extension to internal controllers they do not rely on a specific interface order.

To fill in gaps in the order of interfaces added, a dummy driver `IP_Driver_Dummy` can be added. A sample of such a configuration would be an application that relies on the following order: - IFace0: Internal controller - IFace1: External WiFi module The same hardware might be produced with a different configuration like only providing WiFi but using a cheaper CPU without internal controller. In this case the dummy driver can be used to keep up the same order: - IFace0: Dummy - IFace1: External WiFi module

For drivers and hardware that supports dual Ethernet the requirement to add drivers for internal controllers remain. For using both internal controllers this means: - IFace0: First internal controller - IFace1: Second internal controller - IFace2: External WiFi module When using only the second internal controller the interface index needs to be pushed by using the dummy driver again: - IFace0: Dummy - IFace1: Second internal controller - IFace2: External WiFi module However for using only the first controller of a driver that supports a dual unit, no dummy needs to be added before adding additional external drivers: - IFace0: First internal controller, second is not used. - IFace1: External WiFi module

Additional information

Refer to *Available network interface drivers* on page 580 for a list of available network interface drivers.

Example

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


4.2.3 IP_AddVirtEtherInterface()

Description

Adds a virtual interface to the stack that uses a hardware interface for communication.

Prototype

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

Parameters

Parameter	Description
HWIFaceId	Zero-based interface index of the hardware interface.

Return value

≥ 0 Zero-based interface index of the newly created interface.
< 0 Error.

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.

Optional configuration of the maximum number of interfaces that can be added to the system using `IP_ConfigMaxIFaces()` needs to be done before adding any interface and must not be changed later.

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 interface to the stack.

Prototype

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

Return value

≥ 0	Zero-based interface index of the newly created interface.
< 0	Error.

Additional information

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

Optional configuration of the maximum number of interfaces that can be added to the system using `IP_ConfigMaxIFaces()` needs to be done before adding any interface and must not be changed later.

Example

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


4.2.5 IP_AddMemory()

Description

This function is called from the application to add additional memory to the stack. `IP_AssignMemory()` needs to be called first.

Prototype

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

Parameters

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

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 231 and *`IP_Free`* on page 240.

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(char v);
```

Parameters

Parameter	Description
v	0 to disable, 1 to enable back pressure.

4.2.7 IP_AssignMemory()

Description

Assigns memory to the stack.

Prototype

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

Parameters

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

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_ConfigAgeout()

Description

Configures the timeout for cached ARP entries. The ARP timer removes entries which have not been used for a time longer than AgeOut.

Prototype

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

Parameters

Parameter	Description
Ageout	in Timeout in ms after which an entry is deleted from the ARP cache. Default: 30s.

Additional information

Only effective after adding at least one interface that is capable of using ARP (all kinds of Ethernet interfaces). Might be overwritten if set before adding the first Ethernet interface.

4.2.9 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);
```

Parameters

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.

Additional information

Only effective after adding at least one interface that is capable of using ARP (all kinds of Ethernet interfaces). Might be overwritten if set before adding the first Ethernet interface.

4.2.10 IP_ARP_ConfigAgeoutSniff()

Description

Configures the age out value for ARP entries, which we have created by looking up addresses of received IP packets. The ARP timer removes entries which have not been used for a time longer than AgeOut.

Prototype

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

Parameters

Parameter	Description
Ageout	Timeout in ms after which an entry is deleted from the ARP cache. Default: 500ms.

Additional information

Only effective after adding at least one interface that is capable of using ARP (all kinds of Ethernet interfaces). Might be overwritten if set before adding the first Ethernet interface.

4.2.11 IP_ARP_ConfigAllowGratuitousARP()

Description

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

Prototype

```
void IP_ARP_ConfigAllowGratuitousARP(U8 OnOff);
```

Parameters

Parameter	Description
`OnOff`	Default: On. • 0: Off. • 1: On.

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.

4.2.12 IP_ARP_ConfigAnnounceStaticIP()

Description

Configures whether to announce using a static IP in the network using gratuitous ARP packets.

Prototype

```
void IP_ARP_ConfigAnnounceStaticIP(unsigned IFaceId,
                                   U8       NumAnnouncements);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
NumAnnouncements	Number of gARPs to send.

Additional information

Configures the stack to send a number of gARP packets when a static IP is configured and/or a link-UP for an interface with a static IP set occurs. The gARPs are typically sent with one second delay between them.

A race condition exists between setting a static IP and recognizing a link-UP event that can lead to sending up to twice as many gARPs as configured. As this is not harmful, only occuring very rarely and would need a lot of overhead to prevent this to happen, this should be taken into account when using this feature.

4.2.13 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);
```

Parameters

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

Additional information

Only effective after adding at least one interface that is capable of using ARP (all kinds of Ethernet interfaces). Might be overwritten if set before adding the first Ethernet interface.

4.2.14 IP_ARP_ConfigMaxRetries()

Description

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

Prototype

```
void IP_ARP_ConfigMaxRetries(unsigned Retries);
```

Parameters

Parameter	Description
Retries	Number of retries for sending an ARP request. Default: 8.

Additional information

Only effective after adding at least one interface that is capable of using ARP (all kinds of Ethernet interfaces). Might be overwritten if set before adding the first Ethernet interface.

4.2.15 IP_ARP_ConfigNumEntries()

Description

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

Prototype

```
int IP_ARP_ConfigNumEntries(unsigned MaxNumEntries);
```

Parameters

Parameter	Description
MaxNumEntries	New value to use as number of entries. Default: 8.

Return value

0 OK, stack will try to allocate the requested number of ARP entries.
-1 Error, called after `IP_Init()`.

Additional information

Needs to be called early in `IP_X_Config()`, typically before adding interfaces.

4.2.16 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 BSP_IP_API * pAPI);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pAPI	Pointer to function table to use. For further information regarding BSP_IP_API please refer to Structure BSP_IP_API.

4.2.17 IP_ConfigDoNotAddLowLevelChecks_ARP()

Description

Tells the stack to not add low level ARP checks when initializing the stack with `IP_Init()`.

Prototype

```
void IP_ConfigDoNotAddLowLevelChecks_ARP(void);
```

Additional information

Please refer to `IP_ConfigDoNotAddLowLevelChecks()` for more information.

4.2.18 IP_ConfigDoNotAddLowLevelChecks_UDP()

Description

Tells the stack to not add low level UDP checks when initializing the stack with `IP_Init()`.

Prototype

```
void IP_ConfigDoNotAddLowLevelChecks_UDP(void);
```

Additional information

Please refer to `IP_ConfigDoNotAddLowLevelChecks()` for more information.

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);
```

Parameters

Parameter	Description
NumIFaces	Number of interfaces to allocate memory for.

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);
```

Parameters

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

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);
```

Parameters

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

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);
```

Parameters

Parameter	Description
Off	Offset from cached to uncached area. Can be negative if uncached area is before cached area.

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_ConfigReportSameMacOnNet()

Description

Configures if the stack warns about receiving an Ethernet packet from the same HW address as the interface the packet came in.

Prototype

```
void IP_ConfigReportSameMacOnNet(unsigned   OnOff,
                                 void     * p);
```

Parameters

Parameter	Description
OnOff	• = 0: Disabled. • ≠ 0: Enabled, reports a warning if a duplicate MAC is seen on the network.
p	Reserved for future extensions of this API.

Additional information

The generated warning uses the filter type `IP_MTYPE_APPLICATION` .

4.2.24 IP_ConfigTCPSpace()

Description

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

Prototype

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

Parameters

Parameter	Description
SendSpace	Transmit window size.
RecvSpace	Receive window size.

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.

4.2.25 IP_DisableIPRxChecksum()

Description

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

Prototype

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

Parameters

Parameter	Description
IFace	Zero-based interface index.

Additional information

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

4.2.26 IP_DisableIPv4()

Description

Disables IPv4 in the stack as good as possible.

Prototype

```
void IP_DisableIPv4(void);
```

Additional information

Needs to be called before `IP_Init()` or during `IP_X_Config()` . As IPv4 is a base component of the stack, disabling IPv4 will be done to the best as possible.

Also disables other IPv4 related protocols like ARP and ICMPv4.

4.2.27 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);
```

Parameters

Parameter	Description
pConfig	Pointer to an element of SEGGER_CACHE_CONFIG .
ConfSize	Size of the passed structure in case library and header size of the structure differs.

Additional information

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

4.2.28 IP_DNS_GetServer()

Description

Retrieves the first DNS server configured of the first interface.

Prototype

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

Return value

IP address of the DNS server in host-byte-order.

4.2.29 IP_DNS_GetServerEx()

Description

Retrieves a DNS server configured for an interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
DNSIndex	Zero-based index of the server to retrieve from interface.
pAddr	Pointer to a U32 variable to store the IPv4 DNS address in host-byte-order.
pAddrLen	Length of DNS addr. in bytes. Typically 4 for IPv4.

4.2.30 IP_DNS_ResolveHostEx()

Description

Sends a query to the DNS server. The functions blocks until the reply is received or for a maximum time.

Prototype

```
int IP_DNS_ResolveHostEx(      unsigned           IFaceId,
                         const IP_DNSSD_REQUEST * pRequest,
                               unsigned           ms);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pRequest	Pointer to the request description.
ms	Maximum time to wait for a reply (around 5s for all attempts).

Return value

= 0 Request is valid.
= 1 No reply. All attempts not done.
< 0 No reply all attempts done or request invalid.

4.2.31 IP_DNS_SendDynUpdate()

Description

Build a dynamic update request. It could send the IPv4 address and/or a request to clear all previous records.

Prototype

```
int IP_DNS_SendDynUpdate(      unsigned   IFaceId,
                         const char     * sHost,
                         const char     * sDomain,
                               U32        IPv4Addr,
                               int        ClearPreviousRR,
                               U32        ms);
```

Parameters

Parameter	Description
IFaceId	Index of the interface.
sHost	Null-terminated string of the host to update.
sDomain	Null-terminated string of the domain name.
IPv4Addr	IPv4 address used for the update. Set to 0 to ignore.
ClearPreviousRR	Sent an update request to remove all previous records.
ms	Time in ms that the function is waiting for a reply. The reply might still be fulfilled after the timeout.

Return value

= 1 Send is pending
= 0 Success
< 0 Error

4.2.32 IP_DNS_SetTSIGContext()

Description

Set the TSIG signature context with the parameters needed to perform Secured Dynamic Updates signed with TSIG.

Prototype

```
void IP_DNS_SetTSIGContext
      (char * KeyName,
       char * KeyAlgoName,
       int    ( *pfSign)
(U8 * pData , U16 DataLength , U8 * pDigest , int DigestMaxSize ),
       int    ( *pfGetTime)(U32 * pSeconds ));
```

Parameters

Parameter	Description
KeyName	Pointer to the string containing the key name. Only the pointer is kept so the string must be static.
KeyAlgoName	Pointer to the string containing the algorithm name. Only the pointer is kept so the string must be static.
pfSign	Function pointer on the function which is called to do the crypto signature.
pfGetTime	Function pointer on the function used to get the current time in seconds since 1th January 1970.

4.2.33 IP_DNS_SetMaxTTL()

Description

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

Prototype

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

Parameters

Parameter	Description
TTL	Maximum TTL of a DNS entry in seconds.

Additional information

The real TTL is the minimum of this value and the TTL specified by the DNS server for the entry. The default for the maximum TTL of a DNS entry is 600 seconds.

4.2.34 IP_DNS_SetServer()

Description

Sets the DNS server address of the first interface.

Prototype

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

Parameters

Parameter	Description
DNSServerAddr	IP address of the DNS server.

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.

4.2.35 IP_DNS_SetServerEx()

Description

Sets the IP address of the available DNS servers for an interface.

Prototype

```
int IP_DNS_SetServerEx(      unsigned   IFaceId,
                             U8         DNSIndex,
                       const U8       * pDNSAddr,
                             int        AddrLen);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
DNSIndex	Zero-based index of DNS servers.
pDNSAddr	Pointer to memory location holding the DNS address to set. Typically a 4-byte IP address.
AddrLen	Length of IP address of server. Typically 4-bytes.

Return value

0 OK.
1 - Error.

4.2.36 IP_MDNS_ResolveHost()

Description

Sends a query using Multicast DNS. The functions blocks until the reply is received or for a maximum time.

Prototype

```
int IP_MDNS_ResolveHost(      unsigned           IFaceId,
                        const IP_DNSSD_REQUEST * pRequest,
                              unsigned           ms);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pRequest	Pointer to the request description.
ms	Maximum time [ms] to wait for a reply (around 5s for all attempts).

Return value

= 0 Request is valid.
= 1 No reply. All attempts not done.
< 0 No reply all attempts done or request invalid.

Additional information

When the requested type is A (IPv4 address) or AAAA (IPv6 address), the request is sent for both Apple mDNS and Microsoft LLMNR. Other DNS-SD request are sent only on mDNS.

4.2.37 IP_MDNS_ResolveHostSingleIP()

Description

Sends a query using Multicast DNS. The functions blocks until the reply is received or for a maximum time. Only the first reply is returned, all others will be discarded.

Prototype

```
int IP_MDNS_ResolveHostSingleIP(      unsigned   IFaceId,
                                      void     * pIP,
                                const char     * sHost,
                                      U16        Type,
                                      unsigned   ms);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pIP	Pointer where to store the result. Make sure that enough space is available to store a 4-bytes IPv4 (in host endianness) or 16-bytes IPv6 as requested.
sHost	Hostname to resolve.
Type	Type of desired result: • IP_DNS_SERVER_TYPE_A • IP_DNS_SERVER_TYPE_AAAA
ms	Maximum time [ms] to wait for a reply (around 5s for all attempts).

Return value

= 0 Request is valid.
= 1 No reply. All attempts not done.
< 0 No reply all attempts done or request invalid.

Additional information

The requested is sent for both Apple mDNS and Microsoft LLMNR.

4.2.38 IP_EnableIPRxChecksum()

Description

Enables the IP Rx checksum calculation in the IP header for incoming packets. This is the default behaviour of the stack.

Prototype

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

Parameters

Parameter	Description
IFace	Zero-based interface index.

Additional information

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

4.2.39 IP_GetMaxAvailPacketSize()

Description

Asks the stack for the maximum available free packet size that can then be allocated. (e.g. with a zero-copy alloc).

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index for which the packet shall be allocated.

Return value

No free packet is available at all: 0. Other : Max. packet size that is free.

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.

4.2.40 IP_GetMemPoolInfo()

Description

Collects data about a memory pool such as its size and free bytes.

Prototype

```
int IP_GetMemPoolInfo(void             * pPoolAddr,
                      IP_MEM_POOL_INFO * pInfo);
```

Parameters

Parameter	Description
pPoolAddr	Memory pool to retrieve information for. NULL for the main memory pool added with IP_AssignMemory() .
pInfo	Pointer to structure of IP_MEM_POOL_INFO where to store information about the selected pool.

Return value

= 0 O.K.
≠ 0 Error, memory pool not found ?

4.2.41 IP_GetMTU()

Description

Retrieves the configured TCP MTU size for an interface.

Prototype

```
U32 IP_GetMTU(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

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

4.2.42 IP_GetPrimaryIFace()

Description

Retrieves the currently set primary interface index.

Prototype

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

Return value

Currently set primary interface index. Default is 0.

4.2.43 IP_ICMP_Add()

Description

Adds ICMP Protocol function 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.

4.2.44 IP_ICMP_DisableRxChecksum()

Description

Disables the ICMP Rx checksum calculation. The ICMP checksum computation can be disabled to improve the performance of the stack.

Prototype

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

Parameters

Parameter	Description
IFace	Interface index.

Additional information

In a typical network all data contained in a transferred frame have already been verified by the hardware by checking the trasmitted frames checksum. 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.

4.2.45 IP_ICMP_EnableRxChecksum()

Description

Enables the ICMP Rx checksum calculation. This is the default behaviour of the stack. The ICMP checksum computation can be disabled to improve the performance of the stack.

Prototype

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

Parameters

Parameter	Description
IFace	Zero-based interface index.

Additional information

In a typical network all data contained in a transferred frame have already been verified by the hardware by checking the trasmitted frames checksum. 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.

4.2.46 IP_IGMP_Add()

Description

Adds the IGMP protocol to interface #0.

Prototype

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

Return value

= 0	O.K.
≠ 0	Error.

Additional information

The IGMP (Internet Group Management Protocol) allows a host to JOIN (or subscribe) to a multicast group and receive messages for it. If the switch supports “IGMP snooping” it can then forward multicast packets only to hosts that are subscribed to a group while saving bandwidth on other ports where no host is subscribed to that group. A typical usage example is any form of broadcasting like IPTV where a video feed is sent to a multicast group but switches/routers will only deliver it to the hosts actually interested in receiving the content.

4.2.47 IP_IGMP_AddEx()

Description

Adds the IGMP protocol to an interface.

Prototype

```
int IP_IGMP_AddEx(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

The IGMP (Internet Group Management Protocol) allows a host to JOIN (or subscribe) to a multicast group and receive messages for it. If the switch supports “IGMP snooping” it can then forward multicast packets only to hosts that are subscribed to a group while saving bandwidth on other ports where no host is subscribed to that group. A typical usage example is any form of broadcasting like IPTV where a video feed is sent to a multicast group but switches/routers will only deliver it to the hosts actually interested in receiving the content.

4.2.48 IP_IGMP_ConfigV2AlwaysReport()

Description

Configures if upon every IGMPv2 QUERY a REPORT shall be sent back.

Prototype

```
void IP_IGMP_ConfigV2AlwaysReport(unsigned IFaceId,
                                  U8       OnOff);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	• 0: Default. Do not send a REPORT if another host has already sent a REPORT for a QUERY. • 1: Send a REPORT even if a REPORT from another host has been seen.

Additional information

According to RFC 2236 duplicate REPORTs for the same group shall be avoided. With IGMP snooping on the switch/router this should never happen as REPORTs should not be forwarded anyhow. However there are some faulty switches/routers that forward REPORTs and in such a case we have to respond with a REPORT even if it seems like IGMP snooping is not in use and duplicates should be avoided. Otherwise we might loose our group membership with IGMP snooping due to this faulty implementation on the switch/router.

This behavior can be configured per interface as it might be the case that on a multi interface device one interface is part of a network behaving entirely correct and the other interface being part of a network with faulty switches/routers.

4.2.49 IP_IGMP_JoinGroup()

Description

Joins an IGMP group.

Prototype

```
int IP_IGMP_JoinGroup(unsigned IFaceId,
                      IP_ADDR  GroupIP);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
GroupIP	IGMP group IP to join in host endianness.

Return value

= 0 O.K.
< 0 Error, no memory ?

Additional information

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 switches that will duplicate the packet and send it to everyone that participates the chosen group.

After sending an initial JOIN REPORT the target does not actively participate by sending more unsolicitied messages. The network hardware periodically sends a membership QUERY either to all hosts or specific groups to check that these groups are still in use and if we still want to be part of it.

The “all-systems”/“all-hosts” group 224.0.0.1 is automatically “joined” by opening receive filters for it. This group is a special case as it is a receive only group. In older versions this group had to be joined manually. When calling JOIN for this group it is now ignored and returns O.K.

4.2.50 IP_IGMP_JoinGroup_AutoRejoin()

Description

Joins an IGMP group and rejoins when the interface link state changes. Executed on link DOWN to UP or different in speed/duplex.

Prototype

```
int IP_IGMP_JoinGroup_AutoRejoin(unsigned IFaceId,
                                 IP_ADDR  GroupIP);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
GroupIP	IGMP group IP to join in host endianness.

Return value

= 0 O.K.
< 0 Error, no memory ?

Additional information

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 switches that will duplicate the packet and send it to everyone that participates the chosen group.

After sending an initial JOIN REPORT the target does not actively participate by sending more unsolicitied messages. The network hardware periodically sends a membership QUERY either to all hosts or specific groups to check that these groups are still in use and if we still want to be part of it.

The “all-systems”/“all-hosts” group 224.0.0.1 is automatically “joined” by opening receive filters for it. This group is a special case as it is a receive only group. In older versions this group had to be joined manually. When calling JOIN for this group it is now ignored and returns O.K.

Rejoining groups sends a mesaage immediately after the link change is reported by the system followed by a randomly delayed second message in case the first one got lost (same as for a regular join). To avoid the first message getting lost due to the link change being reported but not immediately being stable/usable, please configure a delay using `IP_ConfigNumLinkUpProbes()` .

Example

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

IP_IGMP_Add();  // IGMP is needed for UPnP
//
// Join the IGMP group for SSDP .
//
IP_IGMP_JoinGroup_AutoRejoin(0, SSDP_IP);
```


4.2.51 IP_IGMP_LeaveGroup()

Description

Leaves an IGMP group.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
GroupIP	IGMP group IP to leave in host endianness.

Example

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

//
// Leave the IGMP group for SSDP .
//
IP_IGMP_LeaveGroup(0, SSDP_IP);
```


4.2.52 IP_RAW_Add()

Description

Adds RAW socket function to 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.

4.2.53 IP_SetAddrMask()

Description

Sets the IP address and subnet mask of an interface. Operates on interface 0.

Prototype

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

Parameters

Parameter	Description
Addr	IP address in host endianness.
Mask	Subnet mask in host endianness.

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 404 for detailed information about the usage of the emNet DHCP client.

Example

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


4.2.54 IP_SetAddrMaskEx()

Description

Sets the IP address and subnet mask of an interface.

Prototype

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

Parameters

Parameter	Description
IFace	Interface number.
Addr	IP address in host endianness.
Mask	Subnet mask in host endianness.

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 404 for detailed information about the usage of the emNet DHCP client.

Example

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


4.2.55 IP_SetGWAddr()

Description

Sets the default gateway address of the selected interface.

Prototype

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

Parameters

Parameter	Description
IFace	Zero-based interface index.
GWAddr	4-byte gateway address in host endianness.

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 404 for detailed information about the usage of the emNet DHCP client.

Example

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


4.2.56 IP_SetHWAddr()

Description

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

Prototype

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

Parameters

Parameter	Description
pHWAddr	6 bytes MAC address.

Additional information

The MAC address needs to be unique for production units.

Example

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


4.2.57 IP_SetHWAddrEx()

Description

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

Prototype

```
void IP_SetHWAddrEx(      unsigned   IFaceId,
                    const U8       * pHWAddr,
                          unsigned   NumBytes);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pHWAddr	Pointer to the MAC address.
NumBytes	Number of bytes of the MAC address (typically 6).

Additional information

The MAC address needs to be unique for production units.

Example

```
IP_SetHWAddrEx(0, "\x00\x22\x33\x44\x55\x66", 6);
```


4.2.58 IP_SetMTU()

Description

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

Prototype

```
void IP_SetMTU(unsigned IFaceId,
               U32      Mtu);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Mtu	Size of maximum transmission unit in bytes.

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.

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.

When being called from `IP_X_Config()` during the configuration phase, the MTU can only be reduced to avoid configuring an MTU bigger than what the interface is capable of. The initial MTU for an interface is set by the stack automatically when an interface is added. After the configuration phase the MTU can freely be set and the application is responsible to make sure to read the initially set MTU using `IP_GetMTU()` and to not configure an MTU higher than that.

Note:

In the supplied emNet 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 emNet* on page 614.

4.2.59 IP_SetRandCallback()

Description

Sets a callback that can provide random data.

Prototype

```
void IP_SetRandCallback(void ( *pfGetRand)(U8 * pBuffer , unsigned NumBytes ));
```

Parameters

Parameter	Description
pfGetRand	Callback function that provides randomized data.

Example

```
/*********************************************************************
*
*       _cbRand()
*
*  Function description
*    Provides a source of randomness.
*
*  Parameters
*    pBuffer : Pointer where to store the random data.
*    NumBytes: Number of random bytes to store.
*/
static void _cbRand(U8* pBuffer, unsigned NumBytes) {
  //
  // Generate NumBytes of random data and store it at pBuffer.
  //
}

IP_SetRandCallback(_cbRand);
```


4.2.60 IP_SetOnIFaceSelectCallback()

Description

Sets a callback that gets notified about an internal interface selection by the stack and allows to override it.

Prototype

```
void IP_SetOnIFaceSelectCallback(IP_ON_IFACE_SELECT_FUNC * pf);
```

Parameters

Parameter	Description
pf	Callback to execute when an interface is selected. Use NULL to remove the callback.

Example

```
/*********************************************************************
*
*  _OnIFaceSelect()
*
*  Function description
*    Callback executed for an internal interface selection. The
*    proposed interface selected internally can be overridden.
*
*  Parameters
*    pFamily: Protocol family (at the moment only PF_INET or PF_INET6).
*    pInfo  : Further information of type IP_ON_IFACE_SELECT_INFO
*             about the interface selection parameters as well as
*             the proposed interface, selected internally based upon
*             these parameters.
*
*  Return value
*    == -1: No suitable interface.
*    >=  0: Interface index to use.
*/
static int _OnIFaceSelect(int PFamily, IP_ON_IFACE_SELECT_INFO* pInfo) {
 //
 // Example: IPv4 firewall out-filter.
 //   Blocking communication with a specific foreign host.
 //   This does not necessarily block communicaton if the
 //   initial transfer was started by the peer as in this
 //   case we might get our interface assigned based on the
 //   interface it came in on. This only causes us to not
 //   find a suitable interface if we do the initial
 //   communication like a connect() to a host.
 //
 if (PFamily == PF_INET) { // We define rules for IPv4 only.
   if (pInfo->pFAddrV4 != NULL) {
     if (htonl(*pInfo->pFAddrV4) == IP_BYTES2ADDR(192, 168, 2, 3)) {
       return -1; // No interface.
     }
   }
 }
 //
 // Do not care about other cases and accept the proposed
 // interface as selected by the stack internally.
 //
 return pInfo->IFaceId;
}

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



```
*  MainTask()
*
*  Function description
*    Main task executed by the RTOS to create further resources and
*    running the main application.
*/
void MainTask(void) {
 IP_Init();
 //
 // Set callback that gets notified when the stack has internally
 // selected an interface.
 //
 IP_SetOnIFaceSelectCallback(_OnIFaceSelect);
 ...
}
```


4.2.61 IP_SetPrimaryIFace()

Description

Sets the primary interface index.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index to use as primary interface of the system. Default is 0.

Return value

= 0 OK.
< 0 Error.

Additional information

The primary interface is given priority for several purposes in multi interface setups. One example would be to use a preferred interface when looking for a DNS server to use in case multiple interface have set DNS servers.

4.2.62 IP_SetSupportedDuplexModes()

Description

Sets the supported duplex/speed of the device to be advertised during Auto-Negotiation.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
DuplexMode	Bitwise-OR combination of the following supported modes: • 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

Return value

= 0 Success
< 0 Not supported.

Additional information

Combining one of the supported duplex/speed modes with IP_PHY_MODE_NO_AUTONEG disables the Auto-Negotiation advertisement and configures a fixed duplex/speed.

4.2.63 IP_SetTTL()

Description

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

Prototype

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

Parameters

Parameter	Description
v	Time-To-Live value.

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.

4.2.64 IP_SetGlobalMcTTL()

Description

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

Prototype

```
void IP_SetGlobalMcTTL(int v);
```

Parameters

Parameter	Description
v	Time-To-Live value.

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.

Global multicast packets are packets with destinations outside the following networks:

- 224.0.0.x
- 239.x.x.x

4.2.65 IP_SetLocalMcTTL()

Description

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

Prototype

```
void IP_SetLocalMcTTL(int v);
```

Parameters

Parameter	Description
v	Time-To-Live value.

Additional information

By default, the TTL (Time-To-Live) is 1. The TTL field length of the IP is 8 bits. The maximum value of the TTL field is therefore 255.

Local multicast packets are packets with destinations inside the following networks:

- 224.0.0.x
- 239.x.x.x

4.2.66 IP_SetUseRxTask()

Description

Sets the internal flag for using the `IP_RxTask()` manually.

Prototype

```
void IP_SetUseRxTask(void);
```

Additional information

The `IP_RxTask` flag has to be set before enabling the interrupt as otherwise it would still be possible for an Rx interrupt to fire before the `IP_RxTask` flag has been set on first execution of said task. Processing the first interrupt(s) without `IP_RxTask` however should not hurt and a device should not be offended by interrupt delay during or directly after init when the task scheduler gets started.

4.2.67 IP_SOCKET_ConfigSelectMultiplicator()

Description

Configures the multiplicator for the timeout parameter of `select()`. Default multiplicator is 1.

Prototype

```
void IP_SOCKET_ConfigSelectMultiplicator(U32 v);
```

Parameters

Parameter	Description
v	Multiplicator to be used.

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 multiplicator 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.

4.2.68 IP_SOCKET_SetDefaultOptions()

Description

Sets the socket options enabled by default.

Prototype

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

Parameters

Parameter	Description
v	Socket options which should be enabled.

Additional information

By default, keepalive (`SO_KEEPALIVE`) socket option is enabled. Refer to `setsockopt()` for a list of supported socket options.

4.2.69 IP_SOCKET_SetLimit()

Description

Sets the maximum number of allowed sockets.

Prototype

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

Parameters

Parameter	Description
Limit	Sets a limit on number of sockets which can be created. The default is 0 which means that no limit is set.

4.2.70 IP_SYSVIEW_Init()

Description

Initializes the profile instrumentation of the stack and SystemView as profiling implementation.

Prototype

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

Additional information

For further information regarding the SysView profiling implementation in emNet please refer to the chapter *Profiling with SystemView* on page 1205.

4.2.71 IP_TCP_Add()

Description

Adds TCP Protocol function 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.

4.2.72 IP_TCP_DisableRxChecksum()

Description

Disables the TCP Rx checksum calculation. The TCP checksum computation can be disabled to improve the performance of the stack.

Prototype

```
void IP_TCP_DisableRxChecksum(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Additional information

In a typical network all data contained in a transferred frame has already been verified by the hardware checking the trasmitted 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.

This only affects the checksum calculation in software. In case the hardware supports TCP Rx checksum calculation it might still discard a received frame in which the TCP checksum is invalid. When supported by hardware, the software calculation is disabled by default and enabled by default if not supported in hardware.

4.2.73 IP_TCP_EnableRxChecksum()

Description

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

Prototype

```
void IP_TCP_EnableRxChecksum(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Additional information

In a typical network all data contained in a transferred frame has already been verified by the hardware checking the trasmitted 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.

This only affects the checksum calculation in software. In case the hardware supports TCP Rx checksum calculation it might still discard a received frame in which the TCP checksum is invalid. When supported by hardware, the software calculation is disabled by default and enabled by default if not supported in hardware.

4.2.74 IP_TCP_Set2MSLDelay()

Description

Sets the maximum segment lifetime (MSL).

Prototype

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

Parameters

Parameter	Description
v	Maximum segment lifetime. The default is 2 seconds.

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.

4.2.75 IP_TCP_SetConnKeepaliveOpt()

Description

Sets the keepalive options.

Prototype

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

Parameters

Parameter	Description
Init	Maximum time [ms] after TCP-connection open (response to SYN) in ms in case no data transfer takes place. The default is IP_TCP_KEEPALIVE_INIT .
Idle	Time [ms] of TCP-inactivity before first keepalive probe is sent. The default is IP_TCP_KEEPALIVE_IDLE .
Period	Time [ms] of TCP-inactivity between keepalive probes. The default is IP_TCP_KEEPALIVE_PERIOD .
MaxRep	Number of keepalive probes before we give up and close the connection. The default is IP_TCP_KEEPALIVE_MAX_REPS repetitions.

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 a one byte keepalive might actually be a 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().

4.2.76 IP_TCP_SetRetransDelayRange()

Description

Sets retransmission delay range.

Prototype

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

Parameters

Parameter	Description
RetransDelayMin	Minimum time [ms] before first retransmission. The default is IP_TCP_RETRANS_MIN . Please note that setting a minimum value below the minimum value of the peer is not recommended and might break delayed ACKs. The default for many stacks is ~200ms, therefore the minimum should be set slightly higher.
RetransDelayMax	Maximum time [ms] to wait before a retransmission. The default is IP_TCP_RETRANS_MAX .

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.

4.2.77 IP_UDP_Add()

Description

Adds UDP Protocol support 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.

4.2.78 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);
```

Parameters

Parameter	Description
LPort	Local port on which to listen for incoming packets.

Return value

≠ NULL O.K. Pointer to the connection.
= NULL Error.

Additional information

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

4.2.79 IP_UDP_DisableRxChecksum()

Description

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

Prototype

```
void IP_UDP_DisableRxChecksum(void);
```

Additional information

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

4.2.80 IP_UDP_EnableRxChecksum()

Description

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

Prototype

```
void IP_UDP_EnableRxChecksum(void);
```

Additional information

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

4.3 Configuration functions (IP fragmentation)

4.3.1 IP_FRAGMENT_ConfigRx()

Description

Modifies the default settings for IPv4 fragmentation.

Prototype

```
void IP_FRAGMENT_ConfigRx(U16 MaxFragments,
                          U32 Timeout,
                          U8  KeepOOO);
```

Parameters

Parameter	Description
MaxFragments	Maximum number of fragments which are allowed for a fragmented packet. Currently 0..255 fragments are allowed.
Timeout	Timeout [ms] before discarding fragment queues.
KeepOOO	Keep Out Of Order fragments. • 0: Discard (default). • 1: Keep.

4.3.2 IP_FRAGMENT_Enable()

Description

Initializes the required variables and adds a timer to the stack to handle outdated fragment queues.

Prototype

```
void IP_FRAGMENT_Enable(void);
```


4.3.3 IP_IPV6_FRAGMENT_ConfigRx()

Description

Modifies the default settings for IPv6 fragmentation.

Prototype

```
void IP_IPV6_FRAGMENT_ConfigRx(U16 MaxFragments,
                               U32 Timeout,
                               U8  KeepOOO);
```

Parameters

Parameter	Description
MaxFragments	Maximum number of fragments which are allowed for a fragmented packet.
Timeout	Timeout [ms] before discarding fragment queues.
KeepOOO	Keep Out Of Order fragments. • 0: Discard (default). • 1: Keep.

4.3.4 IP_IPV6_FRAGMENT_Enable()

Description

Initializes the required variables and adds a timer to the stack to handle outdated fragment queues.

Prototype

```
void IP_IPV6_FRAGMENT_Enable(void);
```


4.4 Management functions

4.4.1 IP_DeInit()

Description

Deinitializes 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 function of the stack called by the application 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
 }
```


4.4.2 IP_Init()

Description

Initializes the TCP/IP stack.

Prototype

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

Return value

= 0	O.K.
< 0	Error.

Additional information

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

Does not detect memory allocation problems during `IP_Init()` at this time. A sufficient memory pool size should be checked by running an `IP_DEBUG` enabled build with `IP_PANIC()` checks first as this will help to discover other problems with the setup as well.

Example

```
#include "IP.h"

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


4.4.3 IP_Task()

Description

Main task for handling the stack.

Prototype

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

Additional information

Implementing this task is the simplest way to include the stack into your project. An example for typical task stack usage is defined by `TASK_STACK_SIZE_IP_TASK` .

For best performance this task should be given a task priority higher than any other IP stack related application task. It however must not have a higher or the same priority than the `IP_RxTask()` or its API alternatives `IP_RXTASK_Init()`, `IP_RXTASK_Exec()` and `IP_RXTASK_WaitForEvent()` .

For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

After startup, this routine settles into a loop, handling received packets and executing other jobs. This loop sleeps until signaled by a driver or a stack internal job being ready for excution.

In case of de-initializing the stack with `IP_DeInit()`, it is possible to leave the loop gracefully by using `IP_ShutDown()` .

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();
}
```


4.4.4 IP_Exec()

Description

Processes received packets and handles timers and other jobs.

Prototype

```
U32 IP_Exec(void);
```

Return value

Value of the next timeout [ms].

Additional information

This function is normally called internally from an endless loop in `IP_Task()` . If no dedicated task running `IP_Task()` is implemented in your project e.g. when using a superloop, `IP_Exec()` should be called regularly.

When being called from a task context, the same task priority rules as for `IP_Task()` apply.

4.4.5 IP_TASK_Init()

Description

Initializes the main IP task context when not using `IP_Task()` .

Prototype

```
void IP_TASK_Init(void);
```

Additional information

The `IP_TASK_*` API is an alternative to using the `IP_Task()` . It allows finer control over the internal steps done in `IP_Task()` . This can be utilized for example to feed a watchdog from the same task periodically.

> **Note**
>
> This routine is not intended to be used when using `IP_Task()` or `IP_Exec()` instead. It needs to be called before `IP_TASK_Exec()` or `IP_TASK_WaitForEvent()` is used.
>
> For best performance the `IP_TASK_*` API should be called with a task priority higher than any other IP stack related application task.

> **Warning**
>
> The task priority from which this routine is executed must not be higher or the same priority than a task executing the `IP_RxTask()` or its API alternatives `IP_RX-TASK_Init()`, `IP_RXTASK_Exec()` and `IP_RXTASK_WaitForEvent()` .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

Example

```
/*********************************************************************
*
*       _IP_Task()
*
*  Function description
*    Application specific implementation of IP_Task() .
*
*  Additional information
*    Allows to insert your own code like feeding a watchdog
*    in-between the separate steps that would be executed by the
*    original task API provided by the stack.
*/
static void _IP_Task(void) {
  unsigned Timeout;

  //
  // Initialize.
  //
  IP_TASK_Init();
  //
  // Task-loop.
  //
  for (;;) {
    //
    // Process received packets and execute pending jobs.
    // The timeout returned is when the next timer-event is due.
    //
    Timeout = IP_TASK_Exec();
```



```
    //
    // Sleep until the next timer-event is due or an event like
    // new packets have been received is signaled.
    //
    IP_TASK_WaitForEvent(Timeout);
  }
}
```


4.4.6 IP_TASK_Exec()

Description

Processes received packets and handles timers and other jobs.

Prototype

```
unsigned IP_TASK_Exec(void);
```

Return value

Value of the next timeout [ms].

Additional information

The `IP_TASK_*` API is an alternative to using the `IP_Task()` . It allows finer control over the internal steps done in `IP_Task()` . This can be utilized for example to feed a watchdog from the same task periodically.

> **Note**
>
> This routine is not intended to be used when using `IP_Task()` or `IP_Exec()` instead.
>
> For best performance the `IP_TASK_*` API should be called with a task priority higher than any other IP stack related application task.

> **Warning**
>
> The task priority from which this routine is executed must not be higher or the same priority than a task executing the `IP_RxTask()` or its API alternatives `IP_RX-TASK_Init()`, `IP_RXTASK_Exec()` and `IP_RXTASK_WaitForEvent()` .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

4.4.7 IP_TASK_WaitForEvent()

Description

Waits for an event for the main IP task to be signaled.

Prototype

```
unsigned IP_TASK_WaitForEvent(unsigned Timeout);
```

Parameters

Parameter	Description
Timeout	Timeout [ms] to wait for an event. 0 for INFINITE is currently not supported (but can be used) and is internally changed to 1 . Typically the timeout value returned by IP_TASK_Exec() should be used.

Return value

= 0 An event was signaled.
≠ 0 Timeout.

Additional information

The IP_TASK_* API is an alternative to using the IP_Task() . It allows finer control over the internal steps done in IP_Task() . This can be utilized for example to feed a watchdog from the same task periodically.

> **Note**
>
> This routine is not intended to be used when using IP_Task() or IP_Exec() instead.
>
> For best performance the IP_TASK_* API should be called with a task priority higher than any other IP stack related application task.

> **Warning**
>
> The task priority from which this routine is executed must not be higher or the same priority than a task executing the IP_RxTask() or its API alternatives IP_RXTASK_Init(), IP_RXTASK_Exec() and IP_RXTASK_WaitForEvent() .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

4.4.8 IP_RxTask()

Description

Optional task to reduce time spent in receive interrupts.

Prototype

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

Additional information

This optional task can be implementing into your project to reduce the time spent in Ethernet receive interrupts. An example for typical task stack usage is defined by `TASK_STACK_SIZE_IP_RX_TASK` .

> **Warning**
>
> This task operates as a pseudo-interrupt executed from task context and is not secured against other API or tasks of the stack. It therefore needs to be given a task priority above all tasks that make use of the API of the stack or one of the other tasks of the stack like the `IP_Task()` or its API alternatives like `IP_TASK_Init()` , `IP_TASK_Exec()` and `IP_TASK_WaitForEvent()` .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

After startup, this routine settles into a loop, receiving/copying packets in a task context instead of from the interrupt itself to reduce interrupt latency. This loop sleeps until signaled by a driver.

In case of de-initializing the stack with `IP_DeInit()`, it is possible to leave the loop gracefully by using `IP_ShutDown()` .

4.4.9 IP_RXTASK_Init()

Description

Initializes the RxTask context when not using `IP_RxTask()` .

Prototype

```
void IP_RXTASK_Init(void);
```

Additional information

The `IP_RXTASK_*` API is an alternative to using the `IP_RxTask()` . It allows finer control over the internal steps done in `IP_RxTask()` . This can be utilized for example to feed a watchdog from the same task periodically.

> **Note**
>
> This routine is not intended to be used when using `IP_RxTask()` instead. It needs to be called before `IP_RXTASK_Exec()` or `IP_RXTASK_WaitForEvent()` is used.

> **Warning**
>
> This routine is part of a pseudo-interrupt executed from task context and is not secured against other API or tasks of the stack. The task priority from which this routine is executed has to be above all tasks that make use of the API of the stack or one of the other tasks of the stack like the `IP_Task()` or its API alternatives like `IP_TASK_Init()`, `IP_TASK_Exec()` and `IP_TASK_WaitForEvent()` .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .
>
> It can however be used from a lower task priority when locking the API with `IP_OS_LOCK()` before and unlocking with `IP_OS_UNLOCK()` after calling this routine.

Example

```
/*********************************************************************
*
*       _IP_RxTask()
*
*  Function description
*    Application specific implementation of IP_RxTask() .
*
*  Additional information
*    Allows to insert your own code like feeding a watchdog
*    in-between the separate steps that would be executed by the
*    original task API provided by the stack.
*/
static void _IP_RxTask(void) {
  //
  // Initialize.
  //
  IP_RXTASK_Init();
  //
  // Task-loop.
  //
  for (;;) {
    //
    // Wait with timeout [ms] (here INFINITE) for the next event to be
    // signaled. Typically the signal is triggered by an interrupt
    // from the driver when receiving new packets.
    //
```



```
    IP_RXTASK_WaitForEvent(0u);
    //
    // Handle received packets and copy them into the stack.
    //
    IP_RXTASK_Exec();
  }
}
```


4.4.10 IP_RXTASK_Exec()

Description

Copies received packets from driver to stack in a task context instead of from an interrupt.

Prototype

```
void IP_RXTASK_Exec(void);
```

Additional information

The `IP_RXTASK_*` API is an alternative to using the `IP_RxTask()` . It allows finer control over the internal steps done in `IP_RxTask()` . This can be utilized for example to feed a watchdog from the same task periodically.

> **Note**
>
> This routine is not intended to be used when using `IP_RxTask()` instead.

> **Warning**
>
> This routine is part of a pseudo-interrupt executed from task context and is not secured against other API or tasks of the stack. The task priority from which this routine is executed has to be above all tasks that make use of the API of the stack or one of the other tasks of the stack like the `IP_Task()` or its API alternatives like `IP_TASK_Init()`, `IP_TASK_Exec()` and `IP_TASK_WaitForEvent()` .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .
>
> It can however be used from a lower task priority when locking the API with `IP_OS_LOCK()` before and unlocking with `IP_OS_UNLOCK()` after calling this routine. In this case you might have to manually remove the `IP_DEBUG` check in the `IP_OS` layer that ensures that the task priorities are used correctly.

4.4.11 IP_RXTASK_WaitForEvent()

Description

Waits for an event for the IP_RxTask to be signaled.

Prototype

```
unsigned IP_RXTASK_WaitForEvent(unsigned Timeout);
```

Parameters

Parameter	Description
Timeout	Timeout [ms] to wait for an event. 0 for INFINITE .

Return value

= 0 An event was signaled.
≠ 0 Timeout.

Additional information

The IP_RXTASK_* API is an alternative to using the IP_RxTask() . It allows finer control over the internal steps done in IP_RxTask() . This can be utilized for example to feed a watchdog from the same task periodically.

Note

This routine is not intended to be used when using IP_RxTask() instead.

Warning

This routine is part of a pseudo-interrupt executed from task context and is not secured against other API or tasks of the stack. The task priority from which this routine is executed has to be above all tasks that make use of the API of the stack or one of the other tasks of the stack like the IP_Task() or its API alternatives like IP_TASK_Init(), IP_TASK_Exec() and IP_TASK_WaitForEvent() .

For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

It can however be used from a lower task priority when locking the API with IP_OS_LOCK() before and unlocking with IP_OS_UNLOCK() after calling this routine. In this case you might have to manually remove the IP_DEBUG check in the IP_OS layer that ensures that the task priorities are used correctly.

4.4.12 IP_Shutdown()

Description

Prepare network stack related tasks for a graceful shutdown.

Prototype

```
unsigned IP_Shutdown(unsigned LeaveTaskLoop,
                     U32      Timeout);
```

Parameters

Parameter	Description
`LeaveTaskLoop`	Leave the task loop(s) of the stack when shutting down the tasks.
`Timeout`	`Timeout` [ms] after which the routine returns regardless of all tasks being able to shut down or not. A timeout of 0 ms for immediate return can be used but the tasks will only be shut down for sure if all of them have a higher priority than the task calling this routine. A non-zero timeout is therefore advised.

Return value

= 0 All tasks have been shut down successfully.
≠ 0 Mask of `IP_TASK_*` bits for the tasks that have not been shut down within the timeout.

Additional information

Before calling `IP_DeInit()` all application tasks should stop calling network API and all tasks that belong directly to the stack like `IP_Task()` should be stopped as well. The later of both is not as easy as the application has no knowledge about the current execution status of these tasks and it might happen that for example `IP_Task()` is currently deep into some protocol like TCP or even deeper like in a callback back into the application.

By calling this routine a graceful stop of these tasks can be requested to prepare them for having their tasks completely removed in the next step.

4.5 Network interface configuration and handling functions

4.5.1 IP_NI_AddPTPDriver()

Description

Adds an NI specific PTP driver for HW timestamp support.

Prototype

```
int IP_NI_AddPTPDriver(      unsigned        IFaceId,
                       const IP_PTP_DRIVER * pPTPDriver,
                             U32             Clock);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pPTPDriver	PTP driver to add.
Clock	Clock [Hz] of the PTP timer. Can not be 0.

Return value

-1 Error, not supported
0 OK
1 Error, called after driver initialization has been completed.

4.5.2 IP_NI_ClrBPressure()

Description

Disables usage of back pressure (sending a jam signal to signal when we run into a shortage where the hardware can not receive more data).

Prototype

```
void IP_NI_ClrBPressure(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

4.5.3 IP_NI_ConfigPoll()

Description

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

Prototype

```
void IP_NI_ConfigPoll(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

4.5.4 IP_NI_ForceCaps()

Description

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

Prototype

```
void IP_NI_ForceCaps(unsigned IFaceId,
                     U8       CapsForcedMask,
                     U8       CapsForcedValue);
```

Parameters

Parameter	Description
IFaceId	Zero-based index network interfaces.
CapsForcedMask	Defines which bits in the Caps byte will be modified. A 1 in the mask will allow the Caps to be modified by the corresponding bit in CapsForcedValue.
CapsForcedValue	Values for the corresponding bits in CapsForcedMask. Usually an OR of the following values: • IP_NI_CAPS_WRITE_IP_CHKSUM • IP_NI_CAPS_WRITE_UDP_CHKSUM • IP_NI_CAPS_WRITE_TCP_CHKSUM • IP_NI_CAPS_WRITE_ICMP_CHKSUM • IP_NI_CAPS_CHECK_IP_CHKSUM • IP_NI_CAPS_CHECK_UDP_CHKSUM • IP_NI_CAPS_CHECK_TCP_CHKSUM • IP_NI_CAPS_CHECK_ICMP_CHKSUM

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);
```


4.5.5 IP_NI_SetBPressure()

Description

Enables usage of back pressure (sending a jam signal to signal when we run into a shortage where the hardware can not receive more data).

Prototype

```
void IP_NI_SetBPressure(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

4.5.6 IP_NI_SetTxBufferSize()

Description

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

Prototype

```
int IP_NI_SetTxBufferSize(unsigned IFaceId,
                          unsigned NumBytes);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
NumBytes	Size of the Tx buffer (at least size of the MTU + 16 bytes for Ethernet).

Return value

-1 Error, not supported
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.

4.6 PHY configuration functions

4.6.1 IP_NI_ConfigPHYAddr()

Description

Configure the PHY Addr.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface number.
Addr	5-bit address.

Additional information

The PHY address is a 5-bit value. The generic PHY driver tries to detect the PHY address automatically, therefore this should not be called if not explicitly needed. If you use this function to set the address explicitly, the function must be called from within `IP_X_Config()`.

4.6.2 IP_NI_ConfigPHYMode()

Description

Configure the PHY mode.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface number.
Mode	The operating mode of the PHY.

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 51 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_Config`* on page 616.

4.6.3 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    pf);
```

Parameters

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.
pf	Callback to PHY config routine.

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);
  ...
}
```


4.6.4 IP_PHY_AddResetHook()

Description

This function adds a hook function to the IP_HOOK_ON_PHY_RESET list. Registered hooks will be called after a PHY reset has been executed and the generic init has been finished. This allows the user to apply further settings to the PHY if needed.

Prototype

```
void IP_PHY_AddResetHook(IP_HOOK_ON_PHY_RESET * pHook,
                         IP_NI_pfOnPhyReset     pf);
```

Parameters

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

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_AddResetHook(&_Hook, _OnPhyReset);  // Register _OnPhyReset() to
                                             // be executed after a PHY
                                             // software reset.
  ...
```



```
}
```


4.6.5 IP_PHY_ConfigAddr()

Description

Configures the PHY address to use.

Prototype

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

Parameters

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

Additional information

New version of the old function `IP_NI_ConfigPHYAddr()` that makes direct use of the PHY module.

4.6.6 IP_PHY_ConfigAfterResetDelay()

Description

Configures the delay between (soft) resetting the PHY and further communication with it.

Prototype

```
void IP_PHY_ConfigAfterResetDelay(unsigned IFaceId,
                                  U16      ms);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
ms	Delay between (soft) resetting the PHY and further communication with it.

4.6.7 IP_PHY_ConfigAltAddr()

Description

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

Prototype

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

Parameters

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

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);
  ...
}
```


4.6.8 IP_PHY_ConfigGigabitSupport()

Description

Configures if the MAC supports Gigabit Ethernet. If the MAC does not support Gigabit Ethernet (default) then the PHY driver does not have to handle it in case it is not supported anyhow.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	• 0: MAC does not support Gigabit Ethernet. • 1: MAC supports Gigabit Ethernet. If the PHY is Gigabit capable as well it can be used.

Additional information

Typically only required if a PHY driver other than the generic PHY driver is used.

4.6.9 IP_PHY_ConfigSupportedModes()

Description

Configures the supported duplex/speed of the device to be advertised during Auto-Negotiation.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Modes	Bitwise-OR combination of the following supported modes: • 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

Additional information

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

Combining one of the supported duplex/speed modes with IP_PHY_MODE_NO_AUTONEG disables the Auto-Negotiation advertisement and configures a fixed duplex/speed.

4.6.10 IP_PHY_ConfigUseStaticFilters()

Description

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

Prototype

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

Parameters

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

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.

4.6.11 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);
```

Parameters

Parameter	Description
Mask	Bitwise-OR bit mask of checks to disable: • PHY_DISABLE_CHECK_ID • PHY_DISABLE_CHECK_LINK_STATE_AFTER_UP • PHY_DISABLE_WATCHDOG

4.6.12 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);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Mask	Bitwise-OR bit mask of checks to disable: • PHY_DISABLE_CHECK_ID • PHY_DISABLE_CHECK_LINK_STATE_AFTER_UP • PHY_DISABLE_WATCHDOG

4.6.13 IP_PHY_ReadReg()

Description

Reads a PHY register.

Prototype

```
int IP_PHY_ReadReg(unsigned   IFaceId,
                   unsigned   RegIndex,
                   unsigned * pData);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
RegIndex	Register index to read.
pData	Pointer where to store the register value read.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

At the moment PHY access is only implemented as a blocking operation including actively waiting for the access to finish. PHY access routines that operate according to the IEEE 802.3 standard have a maximum clock speed of 2.5 MHz. Frequent access to PHY registers might block other operations in the stack.

Example

```
static IP_HOOK_ON_LINK_CHANGE _LinkChangeHook;

static void _OnLinkChange(unsigned IFaceId, U32 Duplex, U32 Speed) {
  unsigned Reg;

  IP_USE_PARA(Duplex);

  if (Speed != 0u) {  // Only on LINK-UP .
    (void)IP_PHY_ReadReg(IFaceId, 0x06u, &Reg);
    if (Reg & 1u) {
      IP_Logf_Application("AutoNeg advertised by peer.");
    } else {
      IP_Logf_Application("No AutoNeg advertised by peer.");
    }
  }
}

/*********************************************************************
*
*       MainTask()
*/
void MainTask(void) {
  IP_Init();
  IP_AddLinkChangeHook(&_LinkChangeHook, _OnLinkChange);
  ...
}
```


4.6.14 IP_AddLinkChangeHook()

Description

Adds a callback that gets executed each time the link state changes.

Prototype

```
void IP_AddLinkChangeHook
      (IP_HOOK_ON_LINK_CHANGE * pHook,
       void                    ( *pf)
(unsigned IFaceId , U32 Duplex , U32 Speed ));
```

Parameters

Parameter	Description
pHook	Management element of type IP_HOOK_ON_LINK_CHANGE.
pf	Callback to execute on a link state change.

Example

```
static IP_HOOK_ON_LINK_CHANGE _Hook;

static void _OnLinkChange(unsigned IFaceId, U32 Duplex, U32 Speed) {
  ...
}

void main(void) {
  ...
  IP_AddLinkChangeHook(&_Hook, _OnLinkChange);  // Register _OnLinkChange() to be

  // executed when interface changes.
  // Connect dial-up interface.
  ...
}
```


4.6.15 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 ));
```

Parameters

Parameter	Description
pHook	Element of type IP_HOOK_ON_PACKET_FREE to register.
pf	Callback that is notified on a packet free.

4.6.16 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 ));
```

Parameters

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

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.
  ...
}
```


4.6.17 IP_PHY_ReInit()

Description

Re-initializes the PHY.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

4.6.18 IP_PHY_SetWdTimeout()

Description

Sets the watchdog timeout for watching if the PHY reached an unstable state.

Prototype

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

Parameters

Parameter	Description
ShiftCnt	Timeout comparison mask is (1 << ShiftCnt) - 1.

Additional information

For optimization reasons the comparison is done by using a bitmask instead of a division. The bitmask is not allowed to contain a zero bit on a lower value position than a one bit. To reach this we pass a shift count instead of a typical timeout.

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.

4.6.19 IP_PHY_WriteReg()

Description

Writes a PHY register.

Prototype

```
int IP_PHY_WriteReg(unsigned IFaceId,
                    unsigned RegIndex,
                    unsigned Data);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
RegIndex	Register index to write.
Data	Data to write to the register.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

At the moment PHY access is only implemented as a blocking operation including actively waiting for the access to finish. PHY access routines that operate according to the IEEE 802.3 standard have a maximum clock speed of 2.5 MHz. Frequent access to PHY registers might block other operations in the stack.

Writes to the PHY registers 0-5 might use an internal caching of the values written. This cache is currently not updated when using this routine and might therefore result in working with wrong values and resetting values when these registers are written by the stack after they have been modified by the application using this routine.

4.7 Statistics functions

4.7.1 IP_STATS_EnableIFaceCounters()

Description

Enables statistic counters for a specific interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7.2 IP_STATS_GetIFaceCounters()

Description

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

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

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.

4.7.3 IP_STATS_GetLastLinkStateChange()

Description

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

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Timestamp in system ticks (typically 1ms).

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7.4 IP_STATS_GetRxBytesCnt()

Description

Retrieves the number of bytes received on an interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Number of bytes received on this interface.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7.5 IP_STATS_GetRxDiscardCnt()

Description

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

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

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.

4.7.6 IP_STATS_GetRxErrCnt()

Description

Retrieves the number of receive errors.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Number of receive errors.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7.7 IP_STATS_GetRxNotUnicastCnt()

Description

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

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

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.

4.7.8 IP_STATS_GetRxUnicastCnt()

Description

Retrieves the number of unicast packets received on an interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Number of unicast packets received on this interface.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7.9 IP_STATS_GetRxUnknownProtoCnt()

Description

Retrieves the number of unknown protocols received.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Number of unknown protocols received.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7.10 IP_STATS_GetTxBytesCnt()

Description

Retrieves the number of bytes sent on an interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Number of bytes sent on this interface.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7.11 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);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

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.

4.7.12 IP_STATS_GetTxErrCnt()

Description

Retrieves the number of send errors on an interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Number of send errors.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.7.13 IP_STATS_GetTxNotUnicastCnt()

Description

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

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

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.

4.7.14 IP_STATS_GetTxUnicastCnt()

Description

Retrieves the number of unicast packets sent on an interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Number of unicast packets sent on this interface.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.8 Other IP Stack functions

4.8.1 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)());
```

Parameters

Parameter	Description
pHook	Pointer to static element of `IP_HOOK_AFTER_INIT` that can be internally used by the stack.
pf	Function pointer to the callback that will be executed.

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();
  ...
}
```


4.8.2 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);
```

Parameters

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.

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);
  ...
}
```


4.8.3 IP_AddInterfaceErrorHook()

Description

Adds a hook function which will be called if initialization fails for an interface.

Prototype

```
void IP_AddInterfaceErrorHook(IP_HOOK_ON_IF_ERROR * pfOnInterfaceError);
```

Parameters

Parameter	Description
pfOnInterfaceError	Pointer to the callback function of type IP_HOOK_ON_IF_ERROR.

4.8.4 IP_AddLinkChangeHook()

Description

Adds a callback that gets executed each time the link state changes.

Prototype

```
void IP_AddLinkChangeHook
      (IP_HOOK_ON_LINK_CHANGE * pHook,
       void                    ( *pf)
(unsigned IFaceId , U32 Duplex , U32 Speed ));
```

Parameters

Parameter	Description
pHook	Management element of type IP_HOOK_ON_LINK_CHANGE.
pf	Callback to execute on a link state change.

Example

```
static IP_HOOK_ON_LINK_CHANGE _Hook;

static void _OnLinkChange(unsigned IFaceId, U32 Duplex, U32 Speed) {
  ...
}

void main(void) {
  ...
  IP_AddLinkChangeHook(&_Hook, _OnLinkChange);  // Register _OnLinkChange() to be

  // executed when interface changes.
  // Connect dial-up interface.
  ...
}
```


4.8.5 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 ));
```

Parameters

Parameter	Description
pHook	Element of type IP_HOOK_ON_PACKET_FREE to register.
pf	Callback that is notified on a packet free.

4.8.6 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 ));
```

Parameters

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

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.
  ...
}
```


4.8.7 IP_Alloc()

Description

Thread safe memory allocation from main IP stack memory pool.

Prototype

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

Parameters

Parameter	Description
NumBytesReq	Number of bytes to allocate.

Return value

= NULL Error.
≠ NULL O.K. Pointer to allocated memory

Additional information

Memory allocated with this function has to be freed with IP_Free().

4.8.8 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);
```

Parameters

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.

Return value

O.K. : Pointer to packet allocated. Error: NULL.

4.8.9 IP_AllocEx()

Description

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

Prototype

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

Parameters

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

Return value

`= NULL` Error.
`≠ NULL` O.K. Pointer to allocated memory

Additional information

Memory allocated with this function has to be freed with `IP_Free()`.

4.8.10 IP_ARP_CleanCache()

Description

Cleans all ARP entries that are not static entries.

Prototype

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


4.8.11 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);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

4.8.12 IP_Connect()

Description

Calls a previously registered hook for the interface if any was set using `IP_SetIFaceConnectHook()`.

Prototype

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

Parameters

Parameter	Description
`IFaceId`	Zero-based interface index.

Return value

= 0 O.K. or no callback set.
< 0 Error.

4.8.13 IP_Disconnect()

Description

Calls a previously registered hook for the interface if any was set using `IP_SetIFaceDisconnectHook()`.

Prototype

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

Parameters

Parameter	Description
`IFaceId`	Zero-based interface index.

Return value

= 0 O.K. or no callback set.
< 0 Error.

4.8.14 IP_Err2Str()

Description

Converts IP stack error code to a readable string by simply using the defines name.

Prototype

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

Parameters

Parameter	Description
x	Error code returned by API of the stack.

Return value

Pointer to string of the define name.

4.8.15 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);
```

Parameters

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

Return value

= -1 Interface not found.
≥ 0 Interface found.

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));
```


4.8.16 IP_Free()

Description

Thread safe memory free to IP stack memory pools.

Prototype

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

Parameters

Parameter	Description
p	Pointer to memory block previously allocated with `IP_Alloc()`.

4.8.17 IP_FreePacket()

Description

Frees a packet back to the stack.

Prototype

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

Parameters

Parameter	Description
pPacket	Packet to free.

Additional information

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

4.8.18 IP_GetAddrMask()

Description

Retrieves the IP address and subnet mask of an interface. The values are stored in host bytes order. (for example, 192.168.1.1 is returned as `0xC0A80101`).

Prototype

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

Parameters

Parameter	Description
`IFace`	Zero-based interface index.
`pAddr`	Address to store the IP address in host order. Can be `NULL`.
`pMask`	Address to store the subnet mask in host order. Can be `NULL`.

4.8.19 IP_GetCurrentLinkSpeed()

Description

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

Prototype

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

Return value

Current link speed in Hertz.

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);
}
```


4.8.20 IP_GetCurrentLinkSpeedEx()

Description

Returns the current link speed of the requested interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface number.

Return value

Current link speed in Hertz.

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);
}
```


4.8.21 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);
```

Parameters

Parameter	Description
NumBytes	Minimum size of packets to find.

Return value

Number of packets available for this size.

4.8.22 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);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Size of header for this interface.

4.8.23 IP_GetGWAddr()

Description

Returns the gateway address of the interface in host endianness. (for example, 192.168.1.1 is returned as `0xc0a80101`).

Prototype

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

Parameters

Parameter	Description
`IFace`	Zero-based interface index.

Return value

The gateway address of the interface.

4.8.24 IP_GetHWAddr()

Description

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

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pDest	Address of the buffer to store the 48-bit MAC address.
Len	Size of the buffer. Should be at least 6-bytes.

4.8.25 IP_GetIPAddr()

Description

Returns the IP address of the interface in host endianness. Example: 192.168.0.1 is returned as `0xc0a80001` for a big endian target, `0x0100a8c0` for a little endian target.

Prototype

```
U32 IP_GetIPAddr(unsigned IFaceId);
```

Parameters

Parameter	Description
`IFaceId`	Zero-based interface number.

Return value

IP address of the interface in host endianness.

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);
}
```


4.8.26 IP_GetIPPacketInfo()

Description

Returns the start address of the data part of an IPv4 packet.

Prototype

```
U8 *IP_GetIPPacketInfo(IP_PACKET * pPacket);
```

Parameters

Parameter	Description
pPacket	Pointer to an IP_PACKET .

Return value

≠ NULL Pointer to the data part of the IPv4 packet.
= NULL Error.

Example

```
/*********************************************************************
*
*       _pfOnRxICMP
*/
static int _pfOnRxICMP(IP_PACKET* pPacket) {
  const U8* 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;
}
```


4.8.27 IP_GetRawPacketInfo()

Description

Returns the start address of the raw data of an IP_PACKET.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to an IP_PACKET structure.
pNumBytes	Output length of the packet.

Return value

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

4.8.28 IP_GetVersion()

Description

Returns the version of the stack.

Prototype

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

Return value

Version number.

Additional information

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

4.8.29 IP_ICMP_AddRxHook()

Description

This function adds a callback that is executed upon receiving an ICMPv4 packet.

Prototype

```
void IP_ICMP_AddRxHook(IP_HOOK_ON_ICMPV4 * pHook,
                       IP_ON_ICMPV4_FUNC * pf,
                       void              * pUserContext);
```

Parameters

Parameter	Description
pHook	Pointer to static element of IP_HOOK_ON_ICMPV4 that can be internally used by the stack.
pf	Function pointer to the callback to execute.
pUserContext	User defined context top pass to the callback.

Example

```
static IP_HOOK_ON_ICMPV4 _Hook;

/*********************************************************************
*
*       _cbOnRx()
*
*  Function description
*    Callback executed when an ICMPv4 packet is received.
*
*  Parameters
*    IFaceId     : Zero-based interface index.
*    pPacket     : Packet that has been received.
*    pUserContext: User context given when adding the hook.
*    p           : Reserved for future extensions of this API.
*
*  Return value
*    == IP_OK                   : Packet has been handled (freed or reused).
*    == IP_OK_TRY_OTHER_HANDLER: Packet is untouched and stack shall try another
 handler.
*
*  Additional information
*    The callback can remove its own hook using IP_ICMP_RemoveRxHook() .
*/
static int _cbOnRx(unsigned   IFaceId,
                   IP_PACKET* pPacket,
                   void*      pUserContext,
                   void*      p) {
  const U8* pData;

  IP_USE_PARA(IFaceId);
  IP_USE_PARA(pUserContext);
  IP_USE_PARA(p);

  pData = IP_GetIPPacketInfo(pPacket);
  if(*pData == IP_ICMP_TYPE_ECHO_REQUEST) {
    IP_Logf_Application("ICMP echo request received!");
  }
  if(*pData == IP_ICMP_TYPE_ECHO_REPLY) {
    IP_Logf_Application("ICMP echo reply received!");
  }
  //
  // Optional: Remove the hook once no longer needed.
```



```
  //
  IP_ICMP_RemoveRxHook(SEGGER_PTR2PTR(IP_HOOK_ON_ICMPV4, pUserContext));
  return IP_OK_TRY_OTHER_HANDLER;  // Let the stack handle the message.
}

/*********************************************************************
*
*  MainTask()
*
*  Function description
*    Main task executed by the RTOS to create further resources and
*    running the main application.
*/
void MainTask(void) {
 IP_Init();
 //
 // Add a hook that gets notified about received ICMP messages.
 // In this example the pointer to the hook item itself is passed as
 // user context to demonstrate the hook removing itself.
 //
 IP_ICMP_AddRxHook(&_Hook, _cbOnRx, &_Hook);
 ...
}
```


4.8.30 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 * pfRxHook);
```

Parameters

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

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 callback is executed AFTER evaluating ICMP replies to our requests but BEFORE answering to foreign requests.

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++);
  }
}
```


4.8.31 IP_ICMP_RemoveRxHook()

Description

This function removes a hook function from the IP_HOOK_ON_ICMPV4 list.

Prototype

```
void IP_ICMP_RemoveRxHook(IP_HOOK_ON_ICMPV4 * pHook);
```

Parameters

Parameter	Description
pHook	Element of type IP_HOOK_ON_ICMPV4 to remove from list.

4.8.32 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. Operates on interface 0.

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);
}
```


4.8.33 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);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface number.

Return value

1 Interface is ready.
0 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);
}
```


4.8.34 IP_IsAllZero()

Description

Checks if there are zeros at the given pointer.

Prototype

```
unsigned IP_IsAllZero(const U8       * p,
                      unsigned   NumBytes);
```

Parameters

Parameter	Description
p	Pointer to location to check for zeros.
NumBytes	Number of bytes to check to be zero.

Return value

0	NOT all bytes are `0x00` at the pointer.
1	All bytes are `0x00` at the pointer.

4.8.35 IP_IsExpired()

Description

Checks if the given system timestamp has already expired.

Prototype

```
unsigned IP_IsExpired(U32 Time);
```

Parameters

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

Return value

1 Time has expired.
0 Time 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);
```


4.8.36 IP_NI_ConfigLinkCheckMultiplier()

Description

Configures the multiplier of the period between interface link checks typically executed each second.

Prototype

```
int IP_NI_ConfigLinkCheckMultiplier(unsigned IFaceId,
                                    unsigned Multiplier);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Multiplier	Multiplier of the link check period (default 1s).

Return value

= 0 O.K.
≠ 0 Error/Not supported.

Additional information

The default period between link checks is one second which is fine for reacting on a link change. For other interfaces like WiFi it might not be necessary to check for the link status each second or it might even be worth reducing link checks to a minimum if this interferes with packet transactions on the same interface like a single SPI.

4.8.37 IP_NI_ConfigUsePromiscuousMode()

Description

Configures if the driver tries to use its hardware precise and hash filters as available before switching to promiscuous mode or if promiscuous mode will be used in any case.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	• 0: Driver will try to use its hardware filters (default). • 1: Driver will be using promiscuous mode.

4.8.38 IP_NI_GetAdminState()

Description

Retrieves the admin state of the given interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

0	Interface disabled.
1	Interface enabled.
-1	Invalid interface ID.

4.8.39 IP_NI_GetIFaceType()

Description

Retrieves a short textual description of the interface type.

Prototype

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

Parameters

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

Return value

= 0 O.K.
≠ 0 Error.

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]);
```


4.8.40 IP_NI_GetState()

Description

Returns the hardware state of the interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

0	Interface is down
1	Interface is up
-1	Invalid interface ID

4.8.41 IP_NI_SetAdminState()

Description

Sets the AdminState of the interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
AdminState	Admin state to set. • 0: DOWN • 1: UP

Additional information

For most interfaces like Ethernet, WiFi and virtual interfaces like VLAN the state is UP by default. Connection oriented interfaces like PPP or PPPoE use the state for a connect request and therefore start with state DOWN.

Setting an interface like Ethernet to DOWN will try to disable this interface to the best possible. A software filter for interfaces with state DOWN discards packets that can not be filtered using other mechanisms.

- Best case: The Rx interrupt of the interface gets disabled, reducing the CPU load for the disabled interface.
- Worst case: Only software filtering is applied. The CPU load for processing incoming packets will remain like for an interface with state UP.

4.8.42 IP_NI_GetTxQueueLen()

Description

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

Prototype

```
int IP_NI_GetTxQueueLen(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

≥ 0 Current Tx queue length.
< 0 Error.

Additional information

IP_SUPPORT_STATS_IFACE or IP_SUPPORT_STATS needs to be enabled.

4.8.43 IP_NI_PauseRx()

Description

Pauses the Rx handling of an interface by disabling it temporary. The Rx handling will be automatically re-enabled after the specified pause time.

Prototype

```
int IP_NI_PauseRx(unsigned IFaceId,
                  U32      Pause);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Pause	Time to pause the Rx handling [ms].

Return value

= 0 O.K.
< 0 Error or disable Rx not supported by driver.

Additional information

Can be called from an interrupt context!

While most of the API is using an API lock that can not be used from an interrupt, this API can be called from an interrupt context as this is the typical case when being flooded with incoming packets. Calling this API from a task might not succeed anymore as the CPU is held constantly busy by Rx.

Unlike `IP_NI_PauseRxInt()` this routine disables the complete Rx logic of the driver which will also prevent being kept busy processing received data from any hardware RxFIFO. By disabling the Rx path completely the RxFIFO no longer is fed.

4.8.44 IP_NI_PauseRxInt()

Description

Pauses the Rx interrupt of an interface by disabling it temporary. The Rx interrupt will be automatically re-enabled after the specified pause time.

Prototype

```
int IP_NI_PauseRxInt(unsigned IFaceId,
                     U32      Pause);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Pause	Time to pause the Rx interrupt [ms].

Return value

= 0 O.K.
< 0 Error or disable Rx interrupt not supported by driver.

Additional information

Can be called from an interrupt context!

While most of the API is using an API lock that can not be used from an interrupt, this API can be called from an interrupt context as this is the typical case when being flooded with incoming packets. Calling this API from a task might not succeed anymore as the CPU is held constantly busy by Rx interrupts.

For plain anti-flooding measurements please use `IP_NI_PauseRx()` .

Pausing the Rx interrupt of an interface can be used as countermeasure to flood situations. It does not prevent the flood of packets being received at the interface itself. It will prevent new Rx interrupts to occur for a certain period of time but will not abort already started Rx handling from a previous interrupt. This means that if the flood keeps on feeding the hardwares RxFIFO for example, it is possible that Rx handling will continue until the hardware is able to read away all incoming data and return to an idle state.

Pausing the Rx interrupt alone but keeping the Rx logic in general enabled can be used to continue receiving incoming data in a flood situation using the `IP_RxTask` while making sure to give Rx a pause once the flooding stops for a moment.

4.8.45 IP_PrintIPAddr()

Description

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

Prototype

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

Parameters

Parameter	Description
pBuffer	Pointer to a buffer where to store the string.
IPAddr	IPv4 addresse in host byte order.
BufferSize	Size of buffer at pBuffer. Should be at least 16 bytes to store xxx.xxx.xxx.xxx .

Return value

> 0 Length of string stored into the buffer without string termination character.
= 0 Buffer is too small.

Additional information

IPAddr is given in host order. Example: 192.168.0.1 is 0xC0A80001 for big endian targets 0x0100A8C0 for little endian targets.

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
}
```


4.8.46 IP_ResolveHost()

Description

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

Prototype

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

Parameters

Parameter	Description
sHost	Host name string to resolve.
pIPAddr	Pointer to where to store the resolved IP addr. in network order.
ms	Timeout in ms to wait for the DNS server to answer.

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.

4.8.47 IP_RemoveEtherTypeHook()

Description

This function removes a hook function for a previously registered Ethernet type.

Prototype

```
void IP_RemoveEtherTypeHook(IP_HOOK_ON_ETH_TYPE * pHook);
```

Parameters

Parameter	Description
pHook	Element of type IP_HOOK_ON_ETH_TYPE to remove.

4.8.48 IP_SendEtherPacket()

Description

Sends a previously allocated Ethernet packet.

Prototype

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

Parameters

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

Return value

= 0 O.K.
≠ 0 Error.

Additional information

The packet gets freed by the stack whether the return code is success or error. The packet can not be reused by the application.

4.8.49 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,
                  unsigned   NumBytes);
```

Parameters

Parameter	Description
`IFace`	Zero-based interface index.
`pData`	Pointer to user data to send.
`NumBytes`	Length of data to send.

Return value

-1 Could not allocate a packet for sending.
0 Packet in out queue.
1 Interface can not send.

4.8.50 IP_SendPing()

Description

Sends a single ICMP echo request (“ping”) to the specified host. Function uses always interface 0.

Prototype

```
int IP_SendPing(U32         FHost,
                char      * pData,
                unsigned    NumBytes,
                U16         SeqNum);
```

Parameters

Parameter	Description
FHost	4-byte IPv4 address in network endian byte order of the target.
pData	Pointer to the ping data, NULL if do not care.
NumBytes	Length of data to attach to ping request.
SeqNum	Ping sequence number.

Return value

= 0 ICMP echo request was successfully sent.
< 0 Error

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 1210 for detailed information about logging.

4.8.51 IP_SendPingCheckReply()

Description

Sends a single ICMP echo request (“ping”) to the specified host using the selected interface and waits for the reply.

Prototype

```
int IP_SendPingCheckReply(U32        IFaceId,
                          U32        FHost,
                          char     * pData,
                          unsigned   NumBytes,
                          unsigned   ms);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
FHost	4-byte IPv4 address in network endian byte order of the target.
pData	Pointer to the ping data, NULL if do not care.
NumBytes	Length of data to attach to ping request.
ms	Number of ms to wait for the reply.

Return value

= 0	OK, ping sent and reply received.
= IP_ERR_TIMEDOUT	Timeout, ping sent but no reply received.
< 0	Error.

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 1210 for detailed information about logging.

4.8.52 IP_SendPingEx()

Description

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

Prototype

```
int IP_SendPingEx(U32        IFaceId,
                  U32        FHost,
                  char     * pData,
                  unsigned   NumBytes,
                  U16        SeqNum);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
FHost	4-byte IPv4 address in network endian byte order of the target.
pData	Pointer to the ping data, NULL if do not care.
NumBytes	Length of data to attach to ping request.
SeqNum	Ping sequence number.

Return value

= 0 OK
< 0 Error

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 1210 for detailed information about logging.

4.8.53 IP_SetIFaceConnectHook()

Description

Sets a hook for an interface that is executed when `IP_Connect()` is called.

Prototype

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

Parameters

Parameter	Description
`IFaceId`	Zero-based interface index.
`pf`	Hook that is called on `IP_Connect()`.

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.

4.8.54 IP_SetIFaceDisconnectHook()

Description

Sets a hook for an interface that is executed when `IP_Disconnect()` is called.

Prototype

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

Parameters

Parameter	Description
`IFaceId`	Zero-based interface index.
`pf`	Hook that is called on `IP_Disconnect()`.

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.

4.8.55 IP_SetOnPacketFreeCallback()

Description

This function sets a callback to be executed once the packet has been freed.

Prototype

```
void IP_SetOnPacketFreeCallback
          (IP_PACKET * pPacket,
           void        ( *pfOnFreeCB)(IP_PACKET * pPacketCB , void * pContextCB ),
           void      * pContext);
```

Parameters

Parameter	Description
pPacket	Pointer to the packet.
pfOnFreeCB	Callback that is notified on a packet free.
pContext	Application context that will be passed to the callback once the packet gets freed.

4.8.56 IP_SetPacketToS()

Description

Sets the value of the ToS/DSCP byte in the IP header of a packet to be sent via the zero-copy API.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to packet buffer.
ToS	ToS byte to use in packet when being sent.

Additional information

The ToS (Type of Service) byte in the IPv4 header has been reused as DSCP (Differentiated Services Code Point) byte with its values remaining somewhat compatible between both use cases.

While the ToS field is only present for IPv4, the DSCP field is present in the same way for IPv4 and IPv6 . If the user intends to explicitly set a ToS value, it is the users responsibility to make sure that he is applying it to an IPv4 packet only.

A good starting point regarding the ToS/DSCP field and its value can be found at the following location: https://en.wikipedia.org/wiki/Type_of_service

4.8.57 IP_SetRxHook()

Description

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

Prototype

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

Parameters

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

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 255 for an example.

4.8.58 IP_SetMicrosecondsCallback()

Description

Sets a callback that is used to get a timestamp in microseconds.

Prototype

```
void IP_SetMicrosecondsCallback(U64 ( *pfGetTime_us)());
```

Parameters

Parameter	Description
pfGetTime_us	The callback to set.

Additional information

Replaces a previously set `IP_SetNanosecondsCallback()` with an internal conversion routine. If your system can provide a nanosecond precise timestamp use `IP_SetNanosecondsCallback()` only.

Example

```
IP_SetMicrosecondsCallback(OS_GetTime_us64);
```


4.8.59 IP_SetNanosecondsCallback()

Description

Sets a callback that is used to get a timestamp in nanoseconds.

Prototype

```
void IP_SetNanosecondsCallback(U64 ( *pfGetTime_ns)());
```

Parameters

Parameter	Description
pfGetTime_ns	The callback to set.

Additional information

Replaces previously set time callbacks of different time bases with an internal conversion as required.

4.9 Stack internal functions, variables and data-structures

emNet internal functions, variables and data-structures are not explained here as they are in no way required to use emNet. 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 emNet. The following data-structures are meant for public usage together with the documented API.

4.9.1 Structure BSP_IP_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;
} BSP_IP_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.

4.9.2 Structure BSP_IP_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, BSP_IP_INSTALL_ISR_PARA* pPara);
  unsigned      (*pfGetMiiMode) (unsigned IFaceId);
  unsigned long (*pfGetEthClock)(void);
} BSP_IP_API;
```

Member	Description
pfInit	Initializes port pins and clocks for Ethernet. Can be NULL.
pfDeInit	De-initializes port pins and clocks for Ethernet. Can be NULL.
pfInstallISR	Installs the driver interrupt handler. Can be NULL. For further information regarding BSP_IP_INSTALL_ISR_PARA please refer to *Structure BSP_IP_INSTALL_ISR_PARA* on page 287.
pfGetMiiMode	Returns the MII mode that the pins have been configured for (0: MII, 1: RMII). Can be NULL.
pfGetEthClock	Returns the clock frequency [Hz] used by the Ethernet peripheral for auto-configuration of internal parameters. Can be NULL.

Additional information

For further information about how this structure is used please refer to *IP_BSP_SetAPI* on page 92.

4.9.3 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
CacheLineSize	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.
pfDMB	Pointer to a callback function that executes a DMB (Data Memory Barrier) instruction to make sure all memory operations are completed. Can be NULL.
pfClean	Pointer to a callback function that executes a clean operation on cached memory. Can be NULL.
pfInvalidate	Pointer to a callback function that executes a clean operation on cached memory. Can be NULL.

Additional information

For further information about how this structure is used please refer to *IP_CACHE_SetConfig* on page 103.

4.9.4 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].

4.9.5 IP_HOOK_ON_IF_ERROR

Description

Callback executed for an error during interface initialization.

Type definition

```
typedef void (IP_HOOK_ON_IF_ERROR)(IP_DRIVER_INTERFACE_ERROR init,
                                   int                       IFaceId,
                                   int                       Errcode);
```

Parameters

Parameter	Description
init	Gives information about which init failed. • init = IP_DRIVER_INTERFACE_INIT_ERROR: Error during driver init. • init = IP_DRIVER_INTERFACE_PHY_ERROR: Error during PHY Init.
IFaceId	Number of interface that failed.
ErrCode	Error Code.

4.9.6 IP_ON_IFACE_SELECT_INFO

Description

Provides information about an internal interface selection for an operation (typically sending without previous receive), as well as to propose an interface.

Type definition

```
typedef struct {
  int               IFaceId;
  const U32       * pLAddrV4;
  const U32       * pFAddrV4;
  const IPV6_ADDR * pLAddrV6;
  const IPV6_ADDR * pFAddrV6;
  U8                Flags;
} IP_ON_IFACE_SELECT_INFO;
```

Structure members

Member	Description
IFaceId	Interface as proposed by internal selection. -1 if no suitable interface was found.
pLAddrV4	Pointer to local IPv4 address. NULL if not used. Value is in network endianness (big endian).
pFAddrV4	Pointer to foreign IPv4 address. NULL if not used. Value is in network endianness (big endian).
pLAddrV6	Pointer to local IPv6 address. NULL if not used.
pFAddrV6	Pointer to foreign IPv6 address. NULL if not used.
Flags	ORR-ed combination of IP_IFACE_SELECT_FLAG_* : • None : Looking for a unicast interface. • IP_ON_IFACE_SELECT_FLAG_BROADCAST: Looking for an interface that is capable of broadcasting. • IP_ON_IFACE_SELECT_FLAG_MULTICAST: Looking for an interface that is capable of multicast.

Additional information

Most parameters are presented as pointers to the actual internal value. If a parameter/pointer is NULL, this means that this parameter was not involved in selecting the proposed interface.

If IFaceId is -1, this means no interface has been selected by the internal procedure.

4.9.7 IP_ON_IFACE_SELECT_FUNC

Description

Callback executed for an internal interface selection. The proposed interface selected internally can be overridden.

Type definition

```
typedef int (IP_ON_IFACE_SELECT_FUNC)(int                       PFamily,
                                      IP_ON_IFACE_SELECT_INFO * pInfo);
```

Parameters

Parameter	Description
PFamily	Protocol family (at the moment only PF_INET or PF_INET6).
pInfo	Further information of type IP_ON_IFACE_SELECT_INFO about the interface selection parameters as well as the proposed interface, selected internally based upon these parameters.

Return value

= -1 No suitable interface.
≥ 0 Interface index to use.

4.9.8 IP_ON_ICMPV4_FUNC

Description

Callback executed when an ICMPv4 packet is received.

Type definition

```
typedef int (IP_ON_ICMPV4_FUNC)(unsigned    IFaceId,
                                 IP_PACKET * pPacket,
                                 void      * pUserContext,
                                 void      * p);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pPacket	Packet that has been received. pPacket->pData points to the IPv4 header.
pUserContext	User context given when adding the hook.
p	Reserved for future extensions of this API.

Return value

IP_OK	Packet has been handled (freed or reused).
IP_OK_TRY_OTHER_HANDLER	Packet is untouched and stack shall try another handler.

Additional information

The callback can remove its own hook using IP_ICMP_RemoveRxHook() .

4.9.9 IP_MEM_POOL_INFO

Description

Provides information about a memory pool managed by the stack.

Type definition

```
typedef struct {
  PTR_ADDR  BaseAddr;
  U32       Size;
  U32       Free;
  U32       MaxFreeChunk;
} IP_MEM_POOL_INFO;
```

Structure members

Member	Description
BaseAddr	Base address of the memory pool.
Size	Total number of bytes managed for this pool.
Free	Number of bytes available for allocation from this pool.
MaxFreeChunk	Biggest chunk available for allocation.

Additional information

Numbers such as free space and maximum chunk size that can be allocated vary by a couple of bytes with values returned being higher than what can be successfully allocated. This is due to internal overhead and alignments that are calculated during allocation processes that are not calculated when retrieving this information about a pool and its free resources.

Chapter 5

Socket interface

The emNet socket API is almost compatible to the Berkeley socket interface. The Berkeley socket interface is the de facto standard for socket communication. emNet 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	
accept()	Accepts an incoming attempt on a socket.
bind()	Assigns a name (port) to an unnamed socket.
closesocket()	Closes a socket.
connect()	Establishes a connection to a socket.
gethostbyname()	Resolve a host name into an IP address.
getpeername()	Fills the passed structure sockaddr with the IP addressing information of the connected host.
getsockname()	Returns the current address to which the socket is bound in the buffer pointed to by pAddr.
getsockopt()	Returns the options associated with a socket.
listen()	Prepares the socket to accept connections.
recv()	Receives data from a connected socket.
recvfrom()	Receives a datagram and stores the source address.
select()	Provides a UNIX-like socket select() call.
send()	Hands data to the stack in order to send it to a connected socket.
sendto()	Hands data to the stack in order to send it to a specified address on a socket.
setsockopt()	Configures some options for the socket.
shutdown()	Stops specific activities on a socket.
socket()	Creates a socket.
emNet specific socket interface functions	
IP_RAW_AddPacketToSocket()	Adds a packet and its data to a RAW socket (buffer).
IP_SOCKET_AbortRead()	Aborts a blocking recv(), recvfrom() and its variations or select() call on a socket.
IP_SOCKET_AddGetSetOptHook()	This function adds a callback that gets executed when the application uses getsockopt()/setsockopt() with the registered option.
IP_SOCKET_CloseAll()	Closes all socket handles that are open.
IP_SOCKET_ConfigSelectMultiplicator()	Configures the multiplicator for the timeout parameter of select().
IP_SOCKET_GetAddrFam()	Returns the IP version of a socket (IPv4 or IPv6).
IP_SOCKET_GetErrorCode()	Returns the last error reported on a socket.
IP_SOCKET_GetLocalPort()	Returns the local port of a socket.
IP_SOCKET_GetNumRxBytes()	Returns the number of received bytes.

Function	Description
IP_SOCKET_SetDefaultOptions()	Sets the socket options enabled by default.
IP_SOCKET_SetLimit()	Sets the maximum number of allowed sockets.
IP_SOCKET_SetLinger()	Activates linger.
IP_SOCKET_SetRxTimeout()	Sets the rx timeout.
IP_SOCK_recvfrom_info()	Receives a datagram and stores the source address and additional information as requested.
IP_SOCK_recvfrom_ts()	Receives a datagram and stores the source address and timestamp.
IP_TCP_Accept()	Registers a callback that will be executed upon a new client.
Set management functions	
IP_FD_CLR()	Removes a socket from a set.
IP_FD_SET()	Adds a socket to a set.
IP_FD_ISSET()	Checks if a socket is part of a set.
Helper macros	
ntohl	Converts a unsigned long value from network to host byte order.
htonl	Converts a unsigned long value from host byte order to network byte order.
htons	Converts a unsigned short value from host byte order to network byte order.
ntohs	Converts a unsigned short value from network to host byte order.

5.1.1 accept()

Description

Accepts an incoming attempt on a socket.

Prototype

```
int accept(int        Socket,
           sockaddr * pSockAddr,
           int      * pAddrLen);
```

Parameters

Parameter	Description
Socket	Socket handle.
pSockAddr	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	As input an optional pointer to a variable with the maximum length of socket address that can be stored. Aa output 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.

Return value

≥ 0 Socket handle of the socket on which the actual connection is made.
= -1 Error.

Additional information

This call is used with connection-based socket types, currently with SOCK_STREAM . Refer to socket() 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 pSockAddr 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 pSockAddr 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 pSockAddr .

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(int        Socket,
         sockaddr * pSockAddr,
         int        AddrLen);
```

Parameters

Parameter	Description
Socket	Socket handle.
pSockAddr	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	The length of the address.

Return value

0	Success.
-1	Error.

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 pSockAddr to the socket.

5.1.3 closesocket()

Description

Closes a socket.

Prototype

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

Parameters

Parameter	Description
Socket	Socket handle.

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 gracefull
*    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(int        Socket,
            sockaddr * pSockAddr,
            int        AddrLen);
```

Parameters

Parameter	Description
Socket	Socket handle.
pSockAddr	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	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.

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 madatory to call IP_RAW_Add() 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 pSockAddr which is an address in the communications space of the socket. Each communications space interprets the pSockAddr 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() for more information), 0 is returned.

You can use the getsockopt() function 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().

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

```
gethostbyname(const char * sName);
```

Parameters

Parameter	Description
sName	Host name to resolve.

Return value

OK: Pointer to a `hostent` structure Error: `NULL` if not successful.

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(int        Socket,
                sockaddr * pSockAddr,
                int      * pAddrLen);
```

Parameters

Parameter	Description
Socket	Socket handle.
pSockAddr	A pointer to a structure of type sockaddr in which the IP address information of the connected host should be stored.
pAddrLen	Max. size of address to return without exceeding the output buffer.

Return value

0 Success.
-1 Error.

Additional information

Refer to *sockaddr* on page 347 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(int        Socket,
                sockaddr * pSockAddr,
                int      * pAddrLen);
```

Parameters

Parameter	Description
Socket	Socket handle.
pSockAddr	A pointer to a structure of type sockaddr in which the IP address information of the connected host should be stored.
pAddrLen	Max. size of address to return without exceeding the output buffer.

Return value

0	Success.
-1	Error.

Additional information

Refer to *sockaddr* on page 347 for detailed information about the structure sockaddr.

5.1.8 getsockopt()

Description

Returns the options associated with a socket.

Prototype

```
int getsockopt(int    Socket,
               int    Level,
               int    Option,
               void * pVal,
               int    ValLen);
```

Parameters

Parameter	Description
Socket	Socket handle.
Level	Compatibility parameter for setsockopt() and getsockopt(). Use symbol SOL_SOCKET.
Option	The socket option which should be retrieved.
pVal	A pointer to the buffer in which the value of the requested option should be stored.
ValLen	The size of the data buffer.

Return value

0 Success.
-1 Error.

Valid values for parameter Option

Value	Description
SO_DONTROUTE	Indicates that outgoing messages must bypass the standard routing facilities.
SO_KEEPALIVE	Indicates that the periodic transmission of messages on a connected socket is enabled. If the connected party fails to respond to these messages, the connection is considered broken. For keepalive behavior configuration please refer to IP_TCP_SetConnKeepaliveOpt().
SO_LINGER	Controls the action taken when unsent messages are queued on a socket and a closesocket() is performed. Refer to closesocket() for detailed information about the linger option.
SO_NOSLOWSTART	Determines if suppressing slow start on this socket is enabled. This option stores an integer value which will contain a non-zero value to suppress slow start or a 0 value to let the socket slow start.
SO_BROADCAST	Determines if sending broadcasts for UDP communication is permitted.
SO_REUSEADDR	Determines if reusing local addresses is allowed when using bind() on a socket. This option stores an integer which will contain a non-zero value to allow reusing addresses or a 0 value to disallow reusing addresses.
SO_RCVTIMEO	Determines the timeout for recv() in ms. A return value of 0 indicates that no timeout is set. This changes the behavior of recv(). recv() is by default a blocking function which

Value	Description
	only returns if data has been received. If a timeout is set `recv()` will return in case of data reception or timeout.
`SO_SNDTIMEO`	Determines the timeout for `send()` in ms. A return value of 0 indicates that no timeout is set. This changes the behavior of `send()`. `send()` is by default a blocking function which only returns if data has been received. If a timeout is set `send()` will return in case of data reception or timeout.
`SO_NONBLOCK`	Determines sockets blocking status. This option stores an integer which will contain a non-zero value to set non-blocking IO or a 0 value to reset non-blocking IO.
`SO_SNDBUF`	Determines the TX buffer size in bytes.
`SO_RCVBUF`	Determines the RX buffer size in bytes.
`SO_MAXMSG`	Determines the Maximum TCP segment size.
`IP_HDRINCL`	Determines if the IP header has to be included by the user for a RAW socket or if the IP header is generated by the stack.
`IP_DONTFRAG`	Determines if fragmentation of large packets is permitted. This option stores an integer value which will contain a non-zero value to suppress fragmentation or a 0 value to allow fragmentation.
`IP_TOS`	Determines the IPv4 type of service.
`IP_TTL`	Determines the IPv4 time to live.
`IP_MULTICAST_TTL`	Determines the IPv4 multicast time to live.
`TCP_MAXSEG`	Determines the maximum segment size in bytes.
`TCP_ACKDELAYTIME`	Determines the time for delayed acks in milliseconds.
`TCP_NOACKDELAY`	Determines if delayed ACKs are suppressed.
`TCP_NODELAY`	Determines if Nagle's Algorithm is disabled. This option stores an integer value which will contain a non-zero value to disable Nagle's Algorithm or a 0 value to enable Nagle's Algorithm.
Read-Only	
`SO_ERROR`	Stores the latest socket error in `pVal` and clears the error in the socket structure.
`SO_MYADDR`	Stores the IP address of the used interface in `pVal`.
`SO_TYPE`	Determines the socket type. For valid socket types refer to
`SO_TXDATA`	Stores the amount of data currently in the TX buffer in bytes.
`SO_RXDATA`	Stores the amount of data currently in the RX buffer in bytes.
Write-Only	
`SO_NBIO`	Sets socket non-blocking status. The specified value will be ignored.
`SO_BIO`	Sets socket blocking status. The specified value will be ignored.
`SO_CALLBACK`	Sets zero-copy callback routine. Refer to *TCP zero-copy interface* on page 354 for detailed information.
`IPV6_UNICAST_HOPS`	Set the unicast hop limit for the socket.
`IPV6_JOIN_GROUP`	Used to join a multicast group on a specified interface.
`IPV6_LEAVE_GROUP`	Used to leave a multicast group on a specified interface.

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 `pVal`. 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 `pVal`. The integer pointed to by `ValLen` should originally contain the size of this buffer; on return, it will be set to the size of the value returned. For `SO_LINGER`, this will be the size of a LINGER structure. For most other options, it will be the size of an integer.

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

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

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

Symbolic name	Value	Description
IP_ERR_FAULT	-25	Generic error for a failed operation.
IP_ERR_NET_UNREACH	-26	No path to the desired network available.
IP_ERR_PARAM	-27	Invalid parameter to function.
IP_ERR_LOGIC	-28	Logical error that should not have happened.
IP_ERR_NOMEM	-29	System error: No memory for requested operation.
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.
IP_ERR_USER_ABORT	-38	When IP_SOCKET_AbortRead() was used on a socket that is currently waiting blocked in recv() or other API.

5.1.9 listen()

Description

Prepares the socket to accept connections.

Prototype

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

Parameters

Parameter	Description
hSock	Socket handle.
Backlog	Backlog for incoming connections. Defines the maximum length of the queue of pending connections.

Return value

-1 Error.
0 Success.

Additional information

The Backlog parameter uses a one by one mapping of its values, meaning the given number means exactly the amount of connections that can be accepted before being processed by calling accept() . As the Backlog parameter is not standardized, other stacks might use different value mappings.

A Backlog parameter of 0 will be increased to 1 as not accepting any connections does not make sense.

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(int    Socket,
         char * pData,
         int    NumBytes,
         int    Flag);
```

Parameters

Parameter	Description
Socket	Handle on the socket.
pData	A pointer to a buffer for incoming data.
NumBytes	Number of bytes of the buffer.
Flag	OR-combination of flags (MSG_PEEK).

Return value

= -1 Error occurred.
= 0 Connection was gracefully closed.
> 0 Number of bytes received.

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() 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(). 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() for more information), -1 is returned.

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

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.

5.1.11 recvfrom()

Description

Receives a datagram and stores the source address.

Prototype

```
int recvfrom(int        Socket,
             char     * pData,
             int        NumBytes,
             int        Flag,
             sockaddr * pFrom,
             int      * pFromLen);
```

Parameters

Parameter	Description
Socket	Handle on the socket.
pData	A pointer to a buffer for incoming data.
NumBytes	Number of bytes of the buffer pointed by pData.
Flag	OR-combination of flags (MSG_PEEK).
pFrom	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. Can be NULL.
pFromLen	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.

Return value

= -1 Error occurred.
≥ 0 Number of bytes received.

Additional information

If pFrom is not a NULL pointer, the source address of the message is filled in. pFromLen is a value-result parameter, initialized to the size of the buffer associated with pFrom , 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() 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() for more information), -1 is returned.

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

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.

5.1.12 select()

Description

Provides a UNIX-like socket `select()` call.

Prototype

```
int select(IP_fd_set * readfds,
           IP_fd_set * writefds,
           IP_fd_set * exceptfds,
           I32         timeout);
```

Parameters

Parameter	Description
`readfds`	Read file descriptor set. Can be `NULL`.
`writefds`	Write file descriptor set. Can be `NULL`.
`exceptfds`	Exception file descriptor set. Can be `NULL`.
`timeout`	Maximum `timeout` [ms] that `select()` should block, waiting for any file descriptor in any given `FD_SET` to become ready. `timeout` of 0 will result in the function retuning immediately. `timeout` of -1 will cause the function to block indefinitely (until one of the descriptors becomes ready or an error occurs).

Return value

= < 0 (`SOCKET_ERROR`)	Error occurred.
= < 0 (`IP_ERR_USER_ABORT`)	Only when calling `IP_SOCKET_AbortRead()` on a socket that is part of the `readfds` .
= 0	Timeout.
> 0	Number of ready file descriptors (sockets) returned over all given descriptor sets.

Additional information

The `select()` call overwrites the given descriptor sets with subsets consisting of those file descriptors (sockets) that are ready. The descriptor sets `readfds`, `writefds` and `exceptfds` may be omitted using `NULL` if no file descriptors are of interest for the specific operation.

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 emNet, socket descriptors are pointers and thus a bit field representation of the descriptor sets is not feasible. Because of this, the emNet 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 emNet 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 emNet to accommodate a larger maximum number of descriptors at the cost of increased processor stack usage.

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

The final difference between the Berkeley and emNet versions of select() is the absence in the emNet 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 emNet 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 322 for detailed information.

Example

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

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

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



```
        socketclose(Socket)
        goto Restart;
      }
    }
    OS_Delay(100);
  }
}
```


5.1.13 send()

Description

Hands data to the stack in order to send it to a connected socket. The stack will copy the data into the socket buffer. In blocking mode, the function returns when all data have been accepted by the stack. If non blocking mode, the function returns immediately.

Prototype

```
int send(      int    Socket,
         const char * pBuffer,
               int    NumBytes,
               int    Flags);
```

Parameters

Parameter	Description
Socket	Socket handle to a connected socket
pBuffer	Pointer to a buffer that contains data to send
NumBytes	Number of bytes to send from pBuffer
Flags	OR-combination of one or more of the valid values listed in the table below.

Return value

< 0	Error (SOCKET_ERROR).
≥ 0	OK, Number of bytes accepted by the stack and ready to be sent. Note: In blocking mode this can only be the full number of bytes, since the function would otherwise block.

Additional information

send() may be used only when the socket is in a connected state. Refer to sendto() 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.

Valid values for parameter Flags

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

5.1.14 sendto()

Description

Hands data to the stack in order to send it to a specified address on a socket. The stack will copy the data into the socket buffer. In blocking mode, the function returns when all data have been accepted by the stack. If non blocking mode, the function returns immediately.

Prototype

```
int sendto(      int        Socket,
           const char     * pBuffer,
                 int        NumBytes,
                 int        Flags,
                 sockaddr * pDestAddr,
                 int        NumBytesAddr);
```

Parameters

Parameter	Description
Socket	Socket handle
pBuffer	Pointer to a buffer that contains data to send
NumBytes	Number of bytes to send from pBuffer
Flags	Ignored at the moment
pDestAddr	Pointer to a buffer containing the destination address
NumBytesAddr	Length of the address stored at pDestAddr in bytes

Return value

< 0	Error (SOCKET_ERROR).
≥ 0	OK, Number of bytes accepted by the stack and ready to be sent. Note: In blocking mode this can only be the full number of bytes, since the function would otherwise block.

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 pDestAddr parameter.

Valid values for parameter Flags

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

5.1.15 setsockopt()

Description

Configures some options for the socket.

Prototype

```
int setsockopt(      int    Socket,
                     int    Level,
                     int    Name,
               const void * pVal,
                     int    ValLen);
```

Parameters

Parameter	Description
Socket	Socket handle.
Level	Level at which the option should be interpreted (SOL_SOCKET for example).
Name	Option enum.
pVal	Pointer on the value for the given option.
ValLen	Length of the data pointed by pVal.

Return value

0 Success.
-1 Error.

Valid values for parameter Option

For valid values and options please refer to getsockopt().

Example

```
void _EnableKeepAlive(long sock) {
    int v = 1;
    setsockopt(sock, SOL_SOCKET, SO_KEEPALIVE, &v, sizeof(v));
}
```


5.1.16 shutdown()

Description

Stops specific activities on a socket.

Prototype

```
int shutdown(int hSock,
             int How);
```

Parameters

Parameter	Description
hSock	Socket handle.
How	One of the following modes: • SHUT_RD : No more receive operations. • SHUT_WR : No more send operations. • SHUT_RDWR: No more receive & send operations.

Return value

0	Success.
-1	SOCKET_ERROR

Additional information

A shutdown() call causes all or a part of a full-duplex connection on the socket associated to be shut down. If How is SHUT_RD, then further receives will be disallowed. If How is SHUT_WR, then further sends will be disallowed. If How is 2, then further receives and sends 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.

Prototype

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

Parameters

Parameter	Description
Domain	Protocol family which should be used.
Type	Specifies the type of the socket.
Proto	Specifies the protocol which should be used with the socket. Must be set to zero except when Type is SOCK_RAW.

Return value

= -1 In case of error.
≥ 0 Socket handle.

Valid values for parameter Domain

Value	Description
AF_INET	IPv4 - Internet protocol version 4
AF_INET6	IPv6 - Internet protocol version 6

Valid values for parameter Type

Value	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_RAW	RAW socket

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 oth-

er 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 128 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 310 and *setsockopt* on page 322 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 322 and requires the user to include his own IP header in the data to send.

5.1.18 IP_RAW_AddPacketToSocket()

Description

Adds a packet and its data to a RAW socket (buffer). The current `pData` pointer and NumBytes of the packet will be used for the payload that will be added to the RAW socket (buffer).

Prototype

```
int IP_RAW_AddPacketToSocket(int         hSock,
                             IP_PACKET * pPacket);
```

Parameters

Parameter	Description
`hSock`	Socket handle of a RAW socket.
`pPacket`	Packet to add to socket (buffer).

Return value

≥ 0 O.K., packet is now handled by the stack.
< 0 Error. Packet has to be freed by application.

Additional information

This function can be used to imitate a packet socket using RAW socket API. As the new data gets stored into the socket buffer without traversing all the way through other layers like IPv4, it is more effective but lacks a proper IP header. Therefore things like getting the source address using `recvfrom()` is not supported and will return invalid data.

5.1.19 IP_SOCKET_AbortRead()

Description

Aborts a blocking `recv()`, `recvfrom()` and its variations or `select()` call on a socket.

Prototype

```
int IP_SOCKET_AbortRead(int hSock);
```

Parameters

Parameter	Description
`hSock`	Socket handler.

Return value

0	O.K.
-1 (`SOCKET_ERROR`)	Error, socket is invalid.

Additional information

Using this routine on a socket that is currently waited on using `recv()` or `recvfrom()` or one of its variations aborts the blocking wait process. The waiting API sets `IP_ERR_USER_ABORT` as socket error. For `select()` instead of `SOCKET_ERROR` , `select()` directly returns `IP_ERR_USER_ABORT` instead of `SOCKET_ERROR` if the socket aborted was part of the "read" `FD_SET` .

5.1.20 IP_SOCKET_AddGetSetOptHook()

Description

This function adds a callback that gets executed when the application uses `getsockopt()`/`setsockopt()` with the registered option.

Prototype

```
void IP_SOCKET_AddGetSetOptHook(IP_SOCK_HOOK_ON_SETGETOPT      * pHook,
                                IP_SOCK_HOOK_ON_GETSETOPT_FUNC * pf,
                                int                              Name);
```

Parameters

Parameter	Description
pHook	Management block of type `IP_SOCK_HOOK_ON_SETGETOPT` .
pf	Callback to execute on `getsockopt()`/`setsockopt()` .
Name	Option name for which the callback gets executed. To avoid conflicts between newly added and existing option names, the base of `IP_SOCK_GETSETOPT_HOOK_NAME_BASE` should be used when implementing your own options.

Example

```
enum {
  APP_SOCK_OPT_VENDOR_NAME = IP_SOCK_GETSETOPT_HOOK_NAME_BASE
};

static char                        _acVendor[32];
static IP_SOCK_HOOK_ON_SETGETOPT  _SockoptHook_VendorName;
static IP_SOCK_HOOK_ON_SETGETOPT  _SockoptHook_SO_RXDATA;
static U8                         _ShowNoteOnce_SO_RXDATA;

/*********************************************************************
*
*       _cbGetSetVendorName()
*
*  Function description
*    Callback for application specific socket option extension of
*    set/get an application specific vendor name.
*
*  Parameters
*    Type  : Source/reason of execution:
*              * IP_SOCK_HOOK_GETOPT
*              * IP_SOCK_HOOK_SETOPT
*    hSock : Socket handle.
*    Level : Socket level such as SOL_SOCKET .
*    Name  : Option name - APP_SOCK_OPT_VENDOR_NAME .
*    pVal  : Pointer to the string to set or where to read to.
*    ValLen: Length of string to set or size of buffer (including
*            string termination) where to read to.
*
*  Return value
*    == IP_SOCK_HOOK_IGNORE_CB: Magic return value used to tell the
*                               stack that while it ended up in the
*                               callback it should still execute its
*                               regular (internal) behavior.
*    == 0                     : O.K.
*    == IP_ERR_LOGIC          : Not able to get the string as no buffer
*                               or a zero buffer has been given.
*    == IP_ERR_MSG_SIZE       : Unable to get/set a string due to string
*                               or buffer size.
*
*/
static int _cbGetSetVendorName(unsigned Type, int hSock, int Level, int Name, void* pVal, int ValLen) {
  unsigned Len;
  int      r;

  IP_USE_PARA(hSock);
  IP_USE_PARA(Level);
  IP_USE_PARA(Name);

  r = 0;  // Assume O.K.
```



```
  if (Type == IP_SOCK_HOOK_GETOPT) {
    //
    // Get vendor name.
    //
    if ((pVal == NULL) || (ValLen == 0)) {
      r = IP_ERR_LOGIC;
    } else {
      Len = strlen(_acVendor) + 1;  // Always count the termination char.
      if ((unsigned)ValLen < Len) {
        r = IP_ERR_MSG_SIZE;
      } else {
        memcpy(pVal, &_acVendor[0], Len);
      }
    }
  } else {
    //
    // Set vendor name.
    //
    if ((pVal == NULL) || (ValLen == 0)) {
      _acVendor[0] = '\0';  // Clear vendor name.
    } else {
      if ((unsigned)ValLen > sizeof(_acVendor)) {
        r = IP_ERR_MSG_SIZE;
      } else {
        memcpy(&_acVendor[0], pVal, ValLen);
      }
    }
  }
  return r;
}

/*********************************************************************
*
*       _cbOnSockopt_SO_RXDATA()
*
*  Function description
*    Callback for application specific socket option extension to
*    notify in case get SO_RXDATA is used.
*
*  Parameters
*    Type  : Source/reason of execution:
*              * IP_SOCK_HOOK_GETOPT
*              * IP_SOCK_HOOK_SETOPT
*    hSock : Socket handle.
*    Level : Socket level such as SOL_SOCKET .
*    Name  : Option name.
*    pVal  : Pointer to the option value.
*    ValLen: Length of the option at pVal .
*
*  Return value
*    == IP_SOCK_HOOK_IGNORE_CB: Magic return value used to tell the
*                               stack that while it ended up in the
*                               callback it should still execute its
*                               regular (internal) behavior.
*    == 0                     : O.K.
*    != 0                     : Error (typically negative) that will be
*                               stored as socket error. The API call
*                               itself will still return SOCKET_ERROR .
*                               Value is limited to the size of signed char.
*
*/
static int _cbOnSockopt_SO_RXDATA(unsigned Type, int hSock, int Level, int Name, void* pVal, int ValLen) {
  IP_USE_PARA(Type);
  IP_USE_PARA(hSock);
  IP_USE_PARA(Level);
  IP_USE_PARA(Name);
  IP_USE_PARA(pVal);
  IP_USE_PARA(ValLen);

  if (_ShowNoteOnce_SO_RXDATA == 0u) {
    _ShowNoteOnce_SO_RXDATA = 1u;
    printf("NOTE: You can use IP_SOCKET_GetNumRxBytes() for a more direct call.\n");
  }
  return IP_SOCK_HOOK_IGNORE_CB;
}

/*********************************************************************
*
*       _InstallAppSocketCallbacks()
*
*  Function description
*    Installs a couple of application specific sample callbacks for
*    getsockopt()/setsockopt() options.
*/
static void _InstallAppSocketCallbacks(void) {
  IP_SOCKET_AddGetSetOptHook(&_SockoptHook_VendorName, _cbGetSetVendorName   , APP_SOCK_OPT_VENDOR_NAME);
```



```
  IP_SOCKET_AddGetSetOptHook(&_SockoptHook_SO_RXDATA , _cbOnSockopt_SO_RXDATA, SO_RXDATA);
}
```


5.1.21 IP_SOCKET_CloseAll()

Description

Closes all socket handles that are open. Can be used to close all sockets in case of changing the local IP address or similar actions that change connection parameters.

Prototype

```
void IP_SOCKET_CloseAll(U32 ConfMask);
```

Parameters

Parameter	Description
ConfMask	Bitwise-OR bit mask of configurations: • CLOSE_ALL_KEEP_LISTEN: Keep listening sockets.

Example: Closing all webserver child tasks

The following code closes all webserver child tasks for example if the target has changed its IP address. The parent listening socket shall be kept open as it is independent from the IP address and typically listens to any address of the system.

```
OS_EnterRegion();  // Avoid being disturbed by disabling task switches.
//
// End all child tasks that might access sockets.
//
for (i = 0; i < MAX_CONNECTIONS; i++) {
  r = OS_IsTask(&_aWebTasks[i]);
  if (r != 0) {
    OS_TerminateTask(&_aWebTasks[i]);
  }
}
//
// Close all sockets that might been abandoned but
// leave open listening parent sockets.
//
IP_SOCKET_CloseAll(CLOSE_ALL_KEEP_LISTEN);
OS_LeaveRegion();  // Allow task switches again.
```


5.1.22 IP_SOCKET_ConfigSelectMultiplicator()

Description

Configures the multiplicator for the timeout parameter of `select()`. Default multiplicator is 1.

Prototype

```
void IP_SOCKET_ConfigSelectMultiplicator(U32 v);
```

Parameters

Parameter	Description
v	Multiplicator to be used.

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 multiplicator 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.

5.1.23 IP_SOCKET_GetAddrFam()

Description

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

Prototype

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

Parameters

Parameter	Description
hSock	Socket handle

Return value

0	Invalid socket handle
AF_INET	IPv4 socket.
AF_INET6	IPv6 socket.

5.1.24 IP_SOCKET_GetErrorCode()

Description

Returns the last error reported on a socket. Returns 0 if the socket has not previously reported an error.

Prototype

```
int IP_SOCKET_GetErrorCode(int hSock);
```

Parameters

Parameter	Description
hSock	Socket handle.

Return value

Last error of the socket. Please refer to the IP.h `IP_ERR_*` return codes for more details.

5.1.25 IP_SOCKET_GetLocalPort()

Description

Returns the local port of a socket.

Prototype

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

Parameters

Parameter	Description
hSock	Socket handle

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.

5.1.26 IP_SOCKET_GetNumRxBytes()

Description

Returns the number of received bytes

Prototype

```
int IP_SOCKET_GetNumRxBytes(int hSock);
```

Parameters

Parameter	Description
hSock	Socket handler.

Return value

> 0	Number of bytes received.
= -1 (SOCKET_ERROR)	Error, socket is invalid.

5.1.27 IP_SOCKET_SetDefaultOptions()

Description

Sets the socket options enabled by default.

Prototype

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

Parameters

Parameter	Description
v	Socket options which should be enabled.

Additional information

By default, keepalive (`SO_KEEPALIVE`) socket option is enabled. Refer to `setsockopt()` for a list of supported socket options.

5.1.28 IP_SOCKET_SetLimit()

Description

Sets the maximum number of allowed sockets.

Prototype

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

Parameters

Parameter	Description
Limit	Sets a limit on number of sockets which can be created. The default is 0 which means that no limit is set.

5.1.29 IP_SOCKET_SetLinger()

Description

Activates linger.

Prototype

```
int IP_SOCKET_SetLinger(int hSock,
                        int Linger);
```

Parameters

Parameter	Description
hSock	Socket handler.
Linger	Flag to activate or deactivate linger.

Return value

0	O.K.
-1 (SOCKET_ERROR)	Error, socket is invalid.

5.1.30 IP_SOCKET_SetRxTimeout()

Description

Sets the rx timeout

Prototype

```
int IP_SOCKET_SetRxTimeout(int hSock,
                           int Timeout);
```

Parameters

Parameter	Description
hSock	Socket handler.
Timeout	New timeout value.

Return value

0	O.K.
-1 (SOCKET_ERROR)	Error, socket is invalid.

5.1.31 IP_SOCK_recvfrom_info()

Description

Receives a datagram and stores the source address and additional information as requested.

Prototype

```
int IP_SOCK_recvfrom_info(int                     hSock,
                          char                  * pData,
                          int                     NumBytes,
                          int                     Flags,
                          sockaddr              * pFrom,
                          int                   * pAddrLen,
                          IP_SOCK_RECVFROM_INFO * pInfo);
```

Parameters

Parameter	Description
hSock	Handle on the socket.
pData	A pointer to a buffer for incoming data.
NumBytes	Number of bytes of the buffer.
Flags	OR-combination of flags (MSG_PEEK).
pFrom	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. Can be NULL.
pAddrLen	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.
pInfo	Pointer where to store additional requested information about the received data.

Return value

= -1 Error occurred.
≥ 0 Number of bytes received.

5.1.32 IP_SOCK_recvfrom_ts()

Description

Receives a datagram and stores the source address and timestamp.

Prototype

```
int IP_SOCK_recvfrom_ts(int                   hSock,
                        char                * pData,
                        int                   NumBytes,
                        int                   Flags,
                        sockaddr            * pFrom,
                        int                 * pAddrLen,
                        IP_PACKET_TIMESTAMP * pTimestamp);
```

Parameters

Parameter	Description
hSock	Handle on the socket.
pData	A pointer to a buffer for incoming data.
NumBytes	Number of bytes of the buffer.
Flags	OR-combination of flags (MSG_PEEK).
pFrom	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. Can be NULL.
pAddrLen	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.
pTimestamp	Pointer where to store the packet timestamp. Can be NULL.

Return value

= -1 Error occurred.
≥ 0 Number of bytes received.

Additional information

Requires IP_SUPPORT_PACKET_TIMESTAMP and/or IP_SUPPORT_PTP to be enabled.

5.1.33 IP_TCP_Accept()

Description

Registers a callback that will be executed upon a new client.

Prototype

```
int IP_TCP_Accept
     (IP_TCP_ACCEPT_HOOK * pHook,
      void                ( *pfAccept)
(int hSock , IP_TCP_ACCEPT_INFO * pInfo , void * pContext ),
      int                 hSock,
      void               * pContext);
```

Parameters

Parameter	Description
pHook	Management element of type IP_TCP_ACCEPT_HOOK.
pfAccept	Callback to register.
hSock	Parent socket handle (needs to have bind() and listen() done).
pContext	Custom context that will be passed to the callback.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

The registered callback has to prevent any blocking situation. Calling send() for example on a non-blocking socket is fine.

Only clients fitting the registered parent socket will be reported to the callback like a regular accept() call. Therefore the usual steps like calling bind() and listen() are still necessary.

5.1.34 IP_FD_CLR()

Description

Removes a socket from a set.

Prototype

```
void IP_FD_CLR(int         hSock,
               IP_fd_set * pSet);
```

Parameters

Parameter	Description
hSock	Socket handle.
pSet	Pointer on a set of type IP_fd_set.

5.1.35 IP_FD_SET()

Description

Adds a socket to a set.

Prototype

```
void IP_FD_SET(int         hSock,
               IP_fd_set * pSet);
```

Parameters

Parameter	Description
hSock	Socket handle.
pSet	Pointer on a set of type IP_fd_set.

5.1.36 IP_FD_ISSET()

Description

Checks if a socket is part of a set.

Prototype

```
int IP_FD_ISSET(int         hSock,
                IP_fd_set * pSet);
```

Parameters

Parameter	Description
hSock	Socket handle.
pSet	Pointer on a set of type IP_fd_set.

Return value

1 Socket is part of the set.
0 Socket is not part of the set.

5.2 Data structures

5.2.1 sockaddr

Description

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

Prototype

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

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

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 *sockaddr_in* on page 348 for more information and an example.

5.2.2 sockaddr_in

Description

Structure for handling Internet addresses.

Prototype

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

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

Example

Refer to *connect* on page 304 for an example.

5.2.3 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.

5.2.4 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.

5.2.5 IP_SOCK_HOOK_ON_GETSETOPT_FUNC

Description

Callback for custom implementations with `setsockopt()`/`getsockopt()`.

Type definition

```
typedef int (IP_SOCK_HOOK_ON_GETSETOPT_FUNC)(unsigned Type,
                                              int      hSock,
                                              int      Level,
                                              int      Name,
                                              void   * pVal,
                                              int      ValLen);
```

Parameters

Parameter	Description
`Type`	Source/reason of execution: • `IP_SOCK_HOOK_GETOPT` • `IP_SOCK_HOOK_SETOPT`
`hSock`	Socket handle.
`Level`	Socket level such as `SOL_SOCKET`.
`Name`	Option name.
`pVal`	Pointer to the option value.
`ValLen`	Length of the option at `pVal`.

Return value

= `IP_SOCK_HOOK_IGNORE_CB`	Magic return value used to tell the stack that while it ended up in the callback it should still execute its regular (internal) behavior.
= 0	O.K.
≠ 0	Error (typically negative) that will be stored as socket error. The API call itself will still return `SOCKET_ERROR`. Value is limited to the size of signed char.

5.2.6 IP_SOCK_RECVFROM_INFO

Description

Returns information about the received UDP packet typically not available with the original BSD compatible call.

Type definition

```
typedef struct {
  IP_PACKET_TIMESTAMP * pTimestamp;
  void                * pLAddrV6;
  unsigned              AddrLenV6;
  U32                   LAddr;
  U8                    IFaceId;
} IP_SOCK_RECVFROM_INFO;
```

Structure members

Member	Description
pTimestamp	Pointer where to store the packet timestamp. Can be NULL.
pLAddrV6	Pointer to buffer where to store the local IPv6 address on which the datagram was received. Can be NULL.
AddrLenV6	Length of the buffer at pLAddrV6 .
LAddr	Local IPv4 address on which the datagram was received.
IFaceId	Zero-based interface index the packet has been received on.

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).

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 TCP 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 for TCP zero-copy

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 for TCP zero-copy

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 for TCP zero-copy

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 TCP zero-copy 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 for TCP zero-copy

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 `IP_ERR_SHUTDOWN` and `IP_ERR_CONN_RESET`. `IP_ERR_SHUTDOWN` defines that the connected peer has closed its end of the connection and sends no more data. `IP_ERR_CONN_RESET` defines that the connected peer has abruptly closed its end of the connection and neither sends nor receives more data.

Returned values

The callback function may return one of the following values:

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

Note: The callback function is called from the stack and is expected to return promptly. No blocking API shall be called from within the callback.

6.3.2 Registering the TCP zero-copy 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);
```

See the function *setsockopt* on page 322 for more details.

6.4 API functions

Function	Description
IP_TCP_Alloc()	Allocates a packet buffer large enough to hold NumBytes bytes of TCP data, plus TCP, IP and MAC headers.
IP_TCP_AllocEx()	Allocates a packet buffer large enough to hold NumBytes bytes of TCP data, plus TCP, IP and MAC headers.
IP_TCP_Free()	Free a packet allocated by IP_TCP_Alloc().
IP_TCP_Send()	Sends a packet buffer on a socket.
IP_TCP_SendAndFree()	Sends a packet buffer on a socket.

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(unsigned NumBytes);
```

Parameters

Parameter	Description
`NumBytes`	Length of the data which should be sent.

Return value

≠ `NULL` Success, pointer to the allocated buffer.
= `NULL` Error.

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()` 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_AllocEx(unsigned NumBytes,
                          unsigned NumBytesHeader);
```

Parameters

Parameter	Description
`NumBytes`	Length of the data which should be sent.
`NumBytesHeader`	Size of all headers (Ethernet + IPvX + TCPvX).

Return value

≠ `NULL` Success, pointer to the allocated buffer.
= `NULL` Error.

Additional information

For further information please refer to `IP_TCP_Alloc()`.

6.4.3 IP_TCP_Free()

Description

Free a packet allocated by `IP_TCP_Alloc()`.

Prototype

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

Parameters

Parameter	Description
p	Pointer to the `IP_Packet` structure.

6.4.4 IP_TCP_Send()

Description

Sends a packet buffer on a socket.

Prototype

```
int IP_TCP_Send(int         hSock,
                IP_PACKET * pPacket);
```

Parameters

Parameter	Description
hSock	Socket handle.
pPacket	Pointer to the IP_PACKET structure.

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 independently 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(int         hSock,
                       IP_PACKET * pPacket);
```

Parameters

Parameter	Description
hSock	Socket handle.
pPacket	Pointer to the IP_PACKET structure.

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 *emNet UDP discover (`IP_UDPDiscover.c` / `IP_UDPDiscover_ZeroCopy.c`)* on page 65 for detailed information about the UDP zero-copy example application.

7.1.1 Allocating, freeing and sending UDP 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 for UDP zero-copy

Applications that use the UDP zero-copy API for receiving data must include a call- back 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 for UDP zero-copy

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 for UDP zero-copy

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 UDP zero-copy 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 for UDP zero-copy

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

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

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

Returned values

The callback function may return one of the following values:

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

Note: The callback function is called from the stack and is expected to return promptly. No blocking API shall be called from within the callback.

7.3.2 Registering the UDP zero-copy 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
*/);
```

For further information, refer to *IP_UDP_Open* on page 381.

7.4 API functions

Function	Description
IP_UDP_Alloc()	Returns a pointer to a packet buffer big enough for the specified sizes.
IP_UDP_AllocEx()	Allocates a packet for UDP on the given interface.
IP_UDP_Close()	Closes a UDP connection handle and removes the connection from demux table list of connections and deallocates it.
IP_UDP_FindFreePort()	Obtains a random port number that is suitable for use as the LPort parameter in a call to IP_UDP_Open().
IP_UDP_Free()	Frees the buffer which was used for a packet.
IP_UDP_GetDataSize()	Returns the size of the data contained in the received UDP packet.
IP_UDP_GetDataPtr()	Returns a pointer to the data contained in the received UDP packet.
IP_UDP_GetDestAddr()	Extracts destination IP address information from a UDP packet.
IP_UDP_GetFPort()	Extracts foreign port information from a UDP packet.
IP_UDP_GetIFIndex()	Extracts the zero-based interface index of the given UDP Packet.
IP_UDP_GetLPort()	Extracts local port information from a UDP packet.
IP_UDP_GetSrcAddr()	Extracts source IP address information from a UDP packet.
IP_UDP_Open()	Creates a UDP connection to receive and pass upwards UDP packets that match the parameters passed.
IP_UDP_OpenEx()	Creates a UDP connection to receive, and pass upwards UDP packets that match the parameters passed.
IP_UDP_Send()	Sends an UDP packet to a specified host.
IP_UDP_SendAndFree()	Sends an UDP packet to a specified host and frees the packet.
IP_UDP_ReducePayloadLen()	Reduces the payload length of an allocated packet.

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(unsigned NumBytesData);
```

Parameters

Parameter	Description
NumBytesData	Length of the data which should be sent.

Return value

≠ NULL Success, pointer to the allocated buffer.
= NULL Error.

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 of the application 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_AllocEx()

Description

Allocates a packet for UDP on the given interface.

Prototype

```
IP_PACKET *IP_UDP_AllocEx(unsigned IFaceId,
                          unsigned NumBytesData);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
NumBytesData	Length of the data which should be sent.

Return value

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

7.4.3 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_CONNECTION * pCon);
```

Parameters

Parameter	Description
pCon	Pointer to the UDP connection.

7.4.4 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()` 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.

7.4.5 IP_UDP_Free()

Description

Frees the buffer which was used for a packet.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

7.4.6 IP_UDP_GetDataSize()

Description

Returns the size of the data contained in the received UDP packet.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

Return value

Size of the data contained in the received UDP packet.

7.4.7 IP_UDP_GetDataPtr()

Description

Returns a pointer to the data contained in the received UDP packet.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

Return value

Pointer to the data part of the UDP packet.

7.4.8 IP_UDP_GetDestAddr()

Description

Extracts destination IP address information from a UDP packet.

Prototype

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

Parameters

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

7.4.9 IP_UDP_GetFPort()

Description

Extracts foreign port information from a UDP packet.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

Return value

Foreign port of the packet.

7.4.10 IP_UDP_GetIFIndex()

Description

Extracts the zero-based interface index of the given UDP Packet.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

Return value

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

7.4.11 IP_UDP_GetLPort()

Description

Extracts local port information from a UDP packet.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

Return value

Local port of the packet.

7.4.12 IP_UDP_GetSrcAddr()

Description

Extracts source IP address information from a UDP packet.

Prototype

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

Parameters

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

7.4.13 IP_UDP_Open()

Description

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

Prototype

```
IP_UDP_CONNECTION *IP_UDP_Open(IP_ADDR   FAddr,
                               U16       FPort,
                               U16       LPort,
                               int       ( *handler)(IP_PACKET * , void * ),
                               void    * pContext);
```

Parameters

Parameter	Description
FAddr	Foreign IP address in network endianness.
FPort	Foreign port in host endianness.
LPort	Local port in host endianness.
handler	Callback function which is called when a UDP packet is received.
pContext	Application defined context pointer.

Return value

≠ NULL Success, pointer to the UDP connection.
= NULL Error.

Additional information

The parameters FAddr, FPort , 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 help 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_OpenEx()

Description

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

Prototype

```
IP_UDP_CONNECTION *IP_UDP_OpenEx(IP_ADDR   FAddr,
                                 U16       FPort,
                                 IP_ADDR   LAddr,
                                 U16       LPort,
                                 int       ( *handler)(IP_PACKET * , void * ),
                                 void    * pContext);
```

Parameters

Parameter	Description
FAddr	Foreign IP address in network endianness.
FPort	Foreign port in host endianness.
LAddr	Local IP address in network endianness.
LPort	Local port in host endianness.
handler	Callback function which is called when a UDP packet is received.
pContext	Application defined context pointer.

Return value

≠ NULL Success, pointer to the UDP connection.
= NULL Error.

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 help 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.

7.4.15 IP_UDP_Send()

Description

Sends an UDP packet to a specified host. Contrarily to `IP_UDP_SendAndFree()`, it does not free the packet in case of error.

Prototype

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

Parameters

Parameter	Description
`IFace`	Zero-based interface index.
`FHost`	IP address of the target host in network endianness.
`fport`	Foreign port in host endianness.
`lport`	Local port in host endianness.
`pPacket`	Data which should be sent to the target host.

Return value

= 0 O.K. Packet sent or in a send FIFO, to be on the wire shortly.
= 1 `IP_ERR_SEND_PENDING`. Packet is waiting for address resolution (incoming ARP response).
< 0 Error code.

Additional information

The packet `pPacket` has to be allocated by calling `IP_UDP_Alloc()`. Refer to `IP_UDP_Alloc()` 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()` for more information about creating a 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 to either free the packet using `IP_UDP_Free()` or to try sending the packet again.

7.4.16 IP_UDP_SendAndFree()

Description

Sends 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);
```

Parameters

Parameter	Description
IFace	Zero-based interface index.
FHost	IP address of the target host in network endianness.
fport	Foreign port in host endianness.
lport	Local port in host endianness.
pPacket	Data which should be sent to the target host.

Return value

= 0 O.K. Packet sent or in a send FIFO, to be on the wire shortly.
= 1 IP_ERR_SEND_PENDING. Packet is waiting for address resolution (incoming ARP response).
< 0 Error code.

Additional information

The packet pPacket has to be allocated by calling IP_UDP_Alloc(). Refer to IP_UDP_Alloc() 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() for more information about creating a UDP connection.

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

7.4.17 IP_UDP_ReducePayloadLen()

Description

Reduces the payload length of an allocated packet.

Prototype

```
int IP_UDP_ReducePayloadLen(IP_PACKET * pPacket,
                            int         NumBytes);
```

Parameters

Parameter	Description
pPacket	Pointer to previously allocated packet.
NumBytes	Reduced payload len.

Return value

< 0 Error, NumBytes parameter is bigger than current len. Other: O.K., current payload len.

Additional information

A previously allocated packet might have been allocated bigger than necessary to be on the safe side. This function allows to reduce the number of bytes that will be sent to the real amount necessary. The payload len can only be reduced. Trying to increase it (again) is returned as error.

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()` during initialization of the stack.

8.1.1 Allocating, freeing and sending packet buffers for RAW Zero-Copy

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 for RAW Zero-Copy

Applications that use the RAW zero-copy API for receiving data must include a call- back 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 for RAW Zero-Copy

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 for RAW Zero-Copy

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

Note: The callback function is called from the stack and is expected to return promptly. No blocking API shall be called from within the callback.

8.3.2 Registering the callback function for RAW Zero-Copy

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 */);
```

See function *IP_RAW_Open* on page 400 for reference.

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
IP_RAW_Alloc()	Returns a pointer to a packet buffer big enough for the specified sizes.
IP_RAW_Close()	Closes a RAW connection handle and removes the connection from demux table list of connections and deallocates it.
IP_RAW_Free()	Frees the buffer which was used for a packet.
IP_RAW_GetDataPtr()	Returns pointer to data contained in the received RAW packet.
IP_RAW_GetDataSize()	Returns size of the payload in the received RAW packet.
IP_RAW_GetDestAddr()	Extracts destination IP address information from a RAW packet.
IP_RAW_GetIFIndex()	Retrieves the zero-based interface index of the given RAW Packet.
IP_RAW_GetSrcAddr()	Extracts source address information from a RAW packet.
IP_RAW_Open()	Creates a RAW connection handle to receive, and pass upwards RAW packets that match the parameters passed.
IP_RAW_Send()	Send a RAW packet to a specified host.
IP_RAW_SendAndFree()	Sends a RAW Packet to a specified host and frees the packet.
IP_RAW_ReducePayloadLen()	Reduces the payload length of an allocated packet.

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,
                        unsigned NumBytesData,
                        int      IpHdrIncl);
```

Parameters

Parameter	Description
IFaceId	Zero-based index of available interfaces.
NumBytesData	Length of the data which should be sent.
IpHdrIncl	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.

Return value

≠ NULL Success, pointer to the allocated buffer.
= NULL Error.

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 of the application 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_CONNECTION * pCon);
```

Parameters

Parameter	Description
pCon	RAW Connection handle.

8.4.3 IP_RAW_Free()

Description

Frees the buffer which was used for a packet.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

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);
```

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

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);
```

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

Return value

Number of data bytes received in the packet.

8.4.6 IP_RAW_GetDestAddr()

Description

Extracts destination IP address information from a RAW packet.

Prototype

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

Parameters

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

8.4.7 IP_RAW_GetIFIndex()

Description

Retrieves the zero-based interface index of the given RAW Packet.

Prototype

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

Parameters

Parameter	Description
pPacket	Pointer to a packet structure.

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);
```

Parameters

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

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_CONNECTION *IP_RAW_Open
                   (IP_ADDR   FAddr,
                    IP_ADDR   LAddr,
                    U8        Protocol,
                    int       ( *handler)(IP_PACKET * pPacket , void * pContext ),
                    void    * pContext);
```

Parameters

Parameter	Description
FAddr	Foreign IP address.
LAddr	Local IP address.
Protocol	IP protocol.
handler	Callback function which is called when a packet of protocol "Protocol" is received.
pContext	in/out Application defined context pointer.

Return value

≠ NULL Success, pointer to the RAW connection handle.
= NULL Error.

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 the reception of any protocol use IPPROTO_RAW as 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 application 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. Contrarily to `IP_RAW_SendAndFree()`, it does not free the packet in case of error.

Prototype

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

Parameters

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

Return value

= 0 O.K. Packet sent or in a send FIFO, to be on the wire shortly.
= 1 `IP_ERR_SEND_PENDING`. Packet is waiting for address resolution (incoming ARP response).
< 0 Error code.

Additional information

The packet `pPacket` has to be allocated by calling `IP_RAW_Alloc()`. Refer to `IP_RAW_Alloc()` 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()` for more information about creating a RAW connection.

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

8.4.11 IP_RAW_SendAndFree()

Description

Sends a RAW Packet to a specified host and frees the packet. Typically called from applications using zero-copy RAW communication. A packet sent with this function is normally allocated by calling `IP_RAW_Alloc()`

Prototype

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

Parameters

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

Return value

= 0 O.K. Packet sent or in a send FIFO, to be on the wire shortly.
= 1 `IP_ERR_SEND_PENDING`. Packet is waiting for address resolution (incoming ARP response).
< 0 Error code.

Additional information

The packet `pPacket` has to be allocated by calling `IP_RAW_Alloc()`. Refer to `IP_RAW_Alloc()` for detailed information.

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

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

8.4.12 IP_RAW_ReducePayloadLen()

Description

Reduces the payload length of an allocated packet.

Prototype

```
int IP_RAW_ReducePayloadLen(IP_PACKET * pPacket,
                            int         NumBytes);
```

Parameters

Parameter	Description
pPacket	Pointer to previously allocated packet.
NumBytes	Reduced payload len.

Return value

< 0 Error, NumBytes parameter is bigger than current len. Other: O.K., current payload len.

Additional information

A previously allocated packet might have been allocated bigger than necessary to be on the safe side. This function allows to reduce the number of bytes that will be sent to the real amount necessary. The payload len can only be reduced. Trying to increase it (again) is returned as error.

Chapter 9

DHCP client

This chapter explains the usage of the Dynamic Host Control Protocol (DHCP) with emNet. 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 assign addresses from a pool. Refer to *[RFC 951] - Bootstrap Protocol* for detailed information about the Bootstrap Protocol.

Most of the `IP_DHCPC_*` API also applies to the `BOOTP` protocol. Therefore no separate API for `BOOTP` is available except for `IP_BOOTPC_Activate()`.

9.2 API functions

Function	Description
IP_BOOTPC_Activate()	Activates the BOOTP client for the specified interface.
IP_DHCPC_Activate()	Activates the DHCP client for the specified interface.
IP_DHCPC_AddStateChangeHook()	This function adds a hook function to the IP_DHCPC_HOOK_ON_STATE_CHANGE list.
IP_DHCPC_AssignCurrentConfig()	Assigns the internally saved configuration received so far to the interface.
IP_DHCPC_ConfigAlwaysStartInit()	Configures if the client always starts with INIT phase, sending a DISCOVER packet, even if an IP was configured for the interface before.
IP_DHCPC_ConfigAssignConfigManually()	Configures if the configuration received by a DHCP server is assigned to the interface as soon as received.
IP_DHCPC_ConfigDisableARPCheck()	Configures if the client checks an offered address to be really free by sending ARP probes before using the IP.
IP_DHCPC_ConfigDNSManually()	Configures if the client will request and use a received DNS server configuration.
IP_DHCPC_ConfigRequestLeaseTime()	Configures the lease time to use in REQUEST messages.
IP_DHCPC_ConfigOnActivate()	Configures behavior regarding currently set parameters of an interface when the DHCP client is activated on this interface.
IP_DHCPC_ConfigOnFail()	Configures behavior regarding currently set parameters of an interface when the DHCP client fails in communication to negotiate a previously received configuration with a server (REQUEST message).
IP_DHCPC_ConfigOnLinkDown()	Configures behavior regarding currently set parameters of an interface when the DHCP client is activated on this interface and the link goes down.
IP_DHCPC_ConfigUniBcStartMode()	Configures if the client will start with unicast or broadcast messages first and enables automatic mode switching.
IP_DHCPC_GetOptionRequestList()	Retrieves the current list of DHCP options to request from a server.
IP_DHCPC_GetState()	Returns the state of the DHCP client.
IP_DHCPC_Halt()	Stops DHCP client activity for the given network interface.
IP_DHCPC_Renew()	Sends a REQUEST with the currently in use DHCP configuration to the DHCP server to check if the configuration is still valid.
IP_DHCPC_SendDeclineAndHalt()	Sends a DECLINE to the DHCP server and halts the DHCP client.
IP_DHCPC_SendDeclineAndResetIP()	Sends a DECLINE to the DHCP server without halting the DHCP client.
IP_DHCPC_SetCallback()	This function allows the caller to set a callback for an interface.

Function	Description
IP_DHCPC_SetClientId()	Sets the DHCP client id for the specified interface.
IP_DHCPC_SetOnOptionCallback()	Sets a callback that gets notified about received DHCP options.
IP_DHCPC_SetOptionRequestList()	Sets the list of DHCP options to request from a server.
IP_DHCPC_SetTimeout()	Sets timeout parameters for DHCP requests.

9.2.1 IP_BOOTPC_Activate()

Description

Activates the BOOTP client for the specified interface.

Prototype

```
int IP_BOOTPC_Activate(int IFaceId);
```

Parameters

Parameter	Description
IFaceId	Interface index.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

9.2.2 IP_DHCPC_Activate()

Description

Activates the DHCP client for the specified interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero based interface index.
sHost	Pointer to host name to use in negotiation. May be NULL.
sDomain	Pointer to domain name to use in negotiation. May be NULL.
sVendor	Pointer to vendor to use in negotiation. May be NULL.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

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

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

Example

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

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

```
DHCP: Sending discover!
DHCP: Received packet from 192.168.1.1
DHCP: Packet type is OFFER.
DHCP: Renewal time: 2160 min.
DHCP: Rebinding time: 3780 min.
DHCP: Lease time: 4320 min.
DHCP: Host name received.
DHCP: Sending Request.
DHCP: Received packet from 192.168.1.1
DHCP: Packet type is ACK.
DHCP: Renewal time: 2160 min.
```



```
DHCP: Rebinding time: 3780 min.
DHCP: Lease time: 4320 min.
DHCP: Host name received.
DHCP: IFace 0: IP: 192.168.199.20, Mask: 255.255.0.0, GW: 192.168.1.1.
```


9.2.3 IP_DHCPC_AddStateChangeHook()

Description

This function adds a hook function to the IP_DHCPC_HOOK_ON_STATE_CHANGE list. Registered hooks will be called with every status change and reports some DHCP informations about the current status.

Prototype

```
void IP_DHCPC_AddStateChangeHook
      (IP_DHCPC_HOOK_ON_STATE_CHANGE * pHook,
       void                           ( *pf)
(unsigned IFaceId , unsigned State , IP_DHCPC_STATE_INFO * pInfo ));
```

Parameters

Parameter	Description
pHook	Element of type IP_DHCPC_HOOK_ON_STATE_CHANGE to register.
pf	Callback that is notified on a state change. • IFaceId: Zero-based interface index. • State : Current DHCP client state. • pInfo : Further information about the current state.

Additional information

This mechanism is provided so that the caller can do some processing when the interface is up (like doing initializations or blinking LEDs, etc.).

The pointer on IP_DHCPC_STATE_INFO structure will not be valid after the callback is called. If parameters are to be used, they need to be copied.

9.2.4 IP_DHCPC_AssignCurrentConfig()

Description

Assigns the internally saved configuration received so far to the interface.

Prototype

```
int IP_DHCPC_AssignCurrentConfig(int IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

1	No configuration available (no previous IP address received).
0	O.K., configuration previously received assigned.
-1	Error, no memory ?

Additional information

Please refer to `IP_DHCPC_ConfigAssignConfigManually()` for more information.

9.2.5 IP_DHCPC_ConfigAlwaysStartInit()

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

```
int IP_DHCPC_ConfigAlwaysStartInit(int IFaceId,
                                   U8  OnOff);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	• 0: Off. • 1: On.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

This function shall be called before activating the DHCP client for an interface using `IP_DHCPC_Activate()` .

9.2.6 IP_DHCPC_ConfigAssignConfigManually()

Description

Configures if the configuration received by a DHCP server is assigned to the interface as soon as received.

Prototype

```
int IP_DHCPC_ConfigAssignConfigManually(int IFaceId,
                                        U8  OnOff);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	• 0: Off (default), configuration is assigned as soon as received. • 1: On, configuration is only saved internally and the user needs to manually assign it to an interface.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

This function shall be called before activating the DHCP client for an interface using `IP_DHCPC_Activate()` .

In case the received configuration shall not be used immediately upon receiving it, it needs to be set manually later on. This can be done by either using information from the state callback using `IP_DHCPC_AddStateChangeHook()` or by simply calling `IP_DHCPC_AssignCurrentConfig()` to activate the configuration as it would have been done automatically.

This configuration does not override assigning a fallback configuration if this has been configured as well.

9.2.7 IP_DHCPC_ConfigDisableARPCheck()

Description

Configures if the client checks an offered address to be really free by sending ARP probes before using the IP.

Prototype

```
int IP_DHCPC_ConfigDisableARPCheck(int IFaceId,
                                   U8  OnOff);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	• 0: Off, ARP probes are sent (default). • 1: On, ARP probes are disabled.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

This function shall be called before activating the DHCP client for an interface using `IP_DHCPC_Activate()`.

This routine is not available when configuring the define `IP_DHCPC_CHECK_IP_BEFORE_BOUND=0`.

9.2.8 IP_DHCPC_ConfigDNSManually()

Description

Configures if the client will request and use a received DNS server configuration.

Prototype

```
int IP_DHCPC_ConfigDNSManually(int IFaceId,
                               U8  OnOff);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	• 0: Off, DNS configuration from server is used. • 1: On, DNS configuration needs to be set manually.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

This function shall be called before activating the DHCP client for an interface using `IP_DHCPC_Activate()` .

9.2.9 IP_DHCPC_ConfigRequestLeaseTime()

Description

Configures the lease time to use in REQUEST messages.

Prototype

```
int IP_DHCPC_ConfigRequestLeaseTime(int IFaceId,
                                    U32 LeaseTime);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
LeaseTime	Lease time [s] to request from the server. The value 0xFFFFFFFF requests an infinte lease from the server. Which lease time is actually granted is decided by the server.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

This function shall be called before activating the DHCP client for an interface using IP_DHCPC_Activate() .

By default the lease time initially granted by the server in its OFFER message is used when sending REQUEST messages. To only initially send a custom lease time you should revert back to a value of 0 (use the previously granted value) or 0xFFFFFFFF (infinity). It is possible to call this routine while the DHCP client is active to change this behavior on the fly.

9.2.10 IP_DHCPC_ConfigOnActivate()

Description

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

Prototype

```
int IP_DHCPC_ConfigOnActivate(int IFaceId,
                              U8  Mode);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Mode	Mode to configure.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

This function shall be called before activating the DHCP client for an interface using IP_DHCPC_Activate() .

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_DHCPC_ConfigOnFail() is configured to use DHCP_RESET_CONFIG (default) it might happen that the static IP will be reset if no server is reachable for the REQUEST or the IP addr. gets declined by a server.

Possible values for Mode

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

9.2.11 IP_DHCPC_ConfigOnFail()

Description

Configures behavior regarding currently set parameters of an interface when the DHCP client fails in communication to negotiate a previously received configuration with a server (REQUEST message).

Prototype

```
int IP_DHCPC_ConfigOnFail(int IFaceId,
                          U8  Mode);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Mode	Mode to configure.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

This function shall be called before activating the DHCP client for an interface using IP_DHCPC_Activate() .

To configure a fallback IP in case no DHCP server is available at all, starting the DHCP client from INIT state, please refer to IP_DHCPC_ConfigOnActivate() .

When the on-fail configuration is applied this does not mean that the DHCP client activity is stopped. It could be intended to keep the DHCP client running in case a server becomes available. To stop the DHCP client you should monitor the state changes using IP_DHCPC_AddStateChangeHook() and react to the messages DHCPC_STATE_INIT and DHCPC_STATE_RESTARTING that signal fallbacks caused by server timeout or no server being available at all. You should then halt the DHCP client service from the callback.

Possible values for Mode

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

9.2.12 IP_DHCPC_ConfigOnLinkDown()

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

```
int IP_DHCPC_ConfigOnLinkDown(int IFaceId,
                              U32 Timeout,
                              U8  Mode);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Timeout	Timeout to wait before reacting on link down [ms].
Mode	Mode to configure.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

This function shall be called before activating the DHCP client for an interface using IP_DHCPC_Activate().

Possible values for Mode

Mode	Description
DHCPC_RESET_CONFIG	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.
DHCPC_USE_STATIC_CONFIG	Setup previous static configuration, if any, as fallback configuration on link down to allow a quick start once the link goes up again.
DHCPC_USE_DHCP_CONFIG	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.

9.2.13 IP_DHCPC_ConfigUniBcStartMode()

Description

Configures if the client will start with unicast or broadcast messages first and enables automatic mode switching.

Prototype

```
int IP_DHCPC_ConfigUniBcStartMode(int IFaceId,
                                  U8  Mode);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Mode	• 0: Start with unicasts first. • 1: Start with broadcasts first.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

Additional information

This function shall be called before activating the DHCP client for an interface using `IP_DHCPC_Activate()` .

The mode switch will be applied after a couple of retries have been sent for the same message. The number of retries can be configured using `IP_DHCPC_SetTimeout()` .

9.2.14 IP_DHCPC_GetState()

Description

Returns the state of the DHCP client.

Prototype

```
unsigned IP_DHCPC_GetState(int IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

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

9.2.15 IP_DHCPC_GetOptionRequestList()

Description

Retrieves the current list of DHCP options to request from a server.

Prototype

```
int IP_DHCPC_GetOptionRequestList(int        IFaceId,
                                  U8       * pBuffer,
                                  unsigned   BufferSize);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pBuffer	Pointer to buffer where to store up to BufferSize DHCP options that are requested from a server. Can be NULL to determine the size of the buffer required to retrieve all options in use.
BufferSize	Maximum amount of options to retrieve.

Return value

< 0	Request list disabled via compile switch or error, no memory ?
≥ 0	Number of U8 DHCP options returned or would be returned if BufferSize would be sufficient. A list with zero entries is valid if it has been set via config.

Additional information

For more information about the actual DHCP options please refer to RFC 1533 . For an example please refer to IP_DHCPC_SetOnOptionCallback() .

9.2.16 IP_DHCPC_Halt()

Description

Stops DHCP client activity for the given network interface.

Prototype

```
int IP_DHCPC_Halt(int IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

9.2.17 IP_DHCPC_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

```
int IP_DHCPC_Renew(int IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

9.2.18 IP_DHCPC_SendDeclineAndHalt()

Description

Sends a DECLINE to the DHCP server and halts the DHCP client.

Prototype

```
int IP_DHCPC_SendDeclineAndHalt(int IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

Please refer to `IP_DHCPC_Decline()` for more information.

9.2.19 IP_DHCPC_SendDeclineAndResetIP()

Description

Sends a DECLINE to the DHCP server without halting the DHCP client. The IP address of the interface is cleared.

Prototype

```
int IP_DHCPC_SendDeclineAndResetIP(int IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

Can be used to reject a previously accepted address from a DHCP server. A reason to do so would be that despite this address seemed free before, now an address collision for example via ACD has been detected. The DHCP client needs to be in BOUND state, otherwise no decline is sent as we do not own the address.

9.2.20 IP_DHCPC_SetCallback()

Description

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

Prototype

```
int IP_DHCPC_SetCallback(int IFaceId,
                         int ( *routine)(int IFaceId , int State ));
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
routine	Callback functions which should be called with every status changes.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

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 initializations or blinking LEDs, etc.).

9.2.21 IP_DHCPC_SetClientId()

Description

Sets the DHCP client id for the specified interface. Should be called prior to `IP_DHCPC_Activate()`

Prototype

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

Parameters

Parameter	Description
`IFaceId`	Zero based interface index.
`pClientId`	Pointer to ClientId to use in negotiation. Will not be copied.
`ClientIdLen`	Length of client ID.

Return value

= 0 O.K.
≠ 0 Error, no memory ?

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.

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_DHCPC_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_DHCPC_SetClientId(0, ClientID, sizeof(ClientID));
```


9.2.22 IP_DHCPC_SetOnOptionCallback()

Description

Sets a callback that gets notified about received DHCP options.

Prototype

```
void IP_DHCPC_SetOnOptionCallback(IP_DHCPC_ON_OPTION_FUNC * pf);
```

Parameters

Parameter	Description
pf	Callback to execute for each DHCP option received.

Example

```
#define DHCP_NTP_OPTION_TYPE  (42u)

static U8 _DhcpReqList[16];  // Default is ~4 U8 options.

/*********************************************************************
*
*       _OnDhcpOption()
*
*  Function description
*    Callback executed for every DHCP option received.
*
*  Parameters
*    IFaceId: Zero-based interface index.
*    pInfo  : Further information of type IP_DHCPC_ON_OPTION_INFO
*             about the DHCP option parsed.
*
*  Additional information
*    Once all options are parsed the end marker (option type 0xFF) is
*    reported as well for an easy to detect end of the list from
*    within the callback. No end is signaled if there was an abort
*    that can be detected by looking at pInfo->Status .
*/
static void _OnDhcpOption(unsigned IFaceId, IP_DHCPC_ON_OPTION_INFO* pInfo) {
  U32 Addr;

  IP_USE_PARA(IFaceId);

  if (pInfo->Status == 0u) {  // Not a parser error ?
    if (pInfo->Type == DHCP_NTP_OPTION_TYPE) {
      //
      // Multiple U32 IPv4 addresses of NTP servers might be returned.
      //
      IP_Logf_Application(  "NTP servers retrieved via DHCP:");
      do {
        //
        // Get the IPv4 address of an NTP server in network endianness (BE)
        // as our printf formatter %i for an IPv4 expects it that way.
        //
        memcpy(&Addr, pInfo->pVal, 4);
        IP_Logf_Application("  - %i", Addr);
        pInfo->Len -= 4u;
      } while (pInfo->Len != 0u);
    }
  }
}

/*********************************************************************
*
*       _AskDhcpForNtpServers()
*
*  Function description
*    When sending a request to a DHCP server, also ask it for NTP servers.
*    To be called from IP_X_Config() before activating the DHCP client.
*
*  Parameters:
*    IFaceId: Zero-based interface index.
*/
static void _AskDhcpForNtpServers(unsigned IFaceId) {
  int NumOptions;

  //
  // Get old (default list).
```



```
  //
  NumOptions = IP_DHCPC_GetOptionRequestList(IFaceId, &_DhcpReqList[0], sizeof(_DhcpReqList));
  if (NumOptions >= 0) {                          // Successfully retrieved current list ?
    if (NumOptions < (int)sizeof(_DhcpReqList)) {  // Do we have space for one more option ?
      //
      // Assume that the NTP option 42 is not in the list and add it.
      // If unsure, add code to look through the options present
      // and only add the option if it is not already in there.
      //
      _DhcpReqList[NumOptions++] = DHCP_NTP_OPTION_TYPE;
      //
      // Set new list.
      //
      IP_DHCPC_SetOptionRequestList(IFaceId, (const U8*)&_DhcpReqList[0], (unsigned)NumOptions);
      //
      // Set callback that gets notified about received options.
      //
      IP_DHCPC_SetOnOptionCallback(_OnDhcpOption);
    }
  }
}
```


9.2.23 IP_DHCPC_SetOptionRequestList()

Description

Sets the list of DHCP options to request from a server.

Prototype

```
int IP_DHCPC_SetOptionRequestList(      int        IFaceId,
                                  const U8       * pOptions,
                                        unsigned   NumOptions);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pOptions	Pointer to array with U8 DHCP options that shall be requested from a server when sending a REQUEST . The memory has to remain valid after the call. Can be NULL for empty list but might prevent the DHCP client from proper functioning.
NumOptions	Number of options at pOptions .

Return value

< 0 Request list disabled via compile switch or error, no memory ?
= 0 O.K.

Additional information

Best practice to add your own DHCP options is to read back the current list of options with IP_DHCPC_GetOptionRequestList() and then add the desired options that are missing.

To set an empty list (whether this makes sense or not) set pOptions ≠ NULL and NumOptions = 0 . For an example please refer to IP_DHCPC_SetOnOptionCallback() .

9.2.24 IP_DHCPC_SetTimeout()

Description

Sets timeout parameters for DHCP requests. RFC2131 demands exponential retransmission times (doubling retransmission time with each retry), but in practice it may make more sense to work with a fixed, non-exponential timeout.

Prototype

```
void IP_DHCPC_SetTimeout(int      IFaceId,
                         U32      Timeout,
                         U32      MaxTries,
                         unsigned Exponential);
```

Parameters

Parameter	Description
IFaceId	Interface index.
Timeout	Value of the timeout [ms].
MaxTries	Maximum number or attempts.
Exponential	Value used to delay new attempts.

9.3 Data structures

9.3.1 IP_DHCPC_ON_OPTION_INFO

Description

Returns information about the next DHCP option to be processed.

Type definition

```
typedef struct {
  const U8 * pVal;
  int        Status;
  U8         Type;
  U8         Len;
} IP_DHCPC_ON_OPTION_INFO;
```

Structure members

Member	Description
pVal	Value of the DHCP option.
Status	• = 0: O.K. • < 0: Parse error, abort of parser.
Type	DHCP option type. Please refer to RFC 1533 for further information.
Len	Length of the option value.

9.3.2 IP_DHCPC_ON_OPTION_FUNC

Description

Callback executed for every DHCP option received.

Type definition

```
typedef void (IP_DHCPC_ON_OPTION_FUNC)(unsigned                  IFaceId,
                                       IP_DHCPC_ON_OPTION_INFO * pInfo);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pInfo	Further information of type IP_DHCPC_ON_OPTION_INFO about the DHCP option parsed.

Additional information

Once all options are parsed, the end marker (option type 0xFF) is reported as well for an easy to detect end of the list from within the callback. No end is signaled if there was an abort that can be detected by looking at pInfo->Status .

Chapter 10

DHCP server (Add-on)

The emNet implementation of the DHCP server is an optional extension to emNet. It allows setting up a Dynamic Host Control Protocol (DHCP) server that seamlessly integrates with emNet. 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 405 in the description of the DHCP client.

10.2 API functions

Function	Description
IP_DHCPS_ConfigDNSAddr()	Configures one or more DNS addr.
IP_DHCPS_ConfigGWAddr()	Configures the default gateway to be assign to clients.
IP_DHCPS_ConfigMaxLeaseTime()	Configures the maximum lease time that a client will be granted to use the achieved configuration.
IP_DHCPS_ConfigPool()	Configures the IP address pool that can be assigned to DHCP clients.
IP_DHCPS_Halt()	Stops DHCP server activity for the passed interface.
IP_DHCPS_Init()	Initializes the DHCP server for the specified interface.
IP_DHCPS_SetReservedAddresses()	Sets a configuration for IP addresses to reserve for specific MAC addresses or HostNames (or both).
IP_DHCPS_SetVendorOptionsCallback()	This function sets a callback that is executed when sending response to a client and the client has sent a vendor class identifier (DHCP option 60).
IP_DHCPS_Start()	Starts the DHCP server for the specified interface.

10.2.1 IP_DHCPS_ConfigDNSAddr()

Description

Configures one or more DNS addr. to assign to clients.

Prototype

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

Parameters

Parameter	Description
IFIndex	Zero-based interface index of the server to configure.
paDNSAddr	Array of U32 IPv4 addresses of DNS servers to use (host order).
NumServers	Number of DNS servers in array.

Return value

0	O.K.
IP_ERR_MISC	Error, server already started.
IP_ERR_PARAM	Error, wrong interface.

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_DHCPS_Start()`.

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_DHCPS_ConfigDNSAddr(0, &aDNSAddr[0], 2);
IP_DHCPS_ConfigPool(0, IP_BYTES2ADDR(192, 168, 12, 11), 0xFFFF0000, 20);
IP_DHCPS_Init(0);
IP_DHCPS_Start(0);
```


10.2.2 IP_DHCPS_ConfigGWAddr()

Description

Configures the default gateway to be assign to clients.

Prototype

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

Parameters

Parameter	Description
IFIndex	Zero-based interface index of the server to configure.
GWAddr	Default gateway IP address in host order.

Return value

0	O.K.
IP_ERR_MISC	Error, server already started.
IP_ERR_PARAM	Error, wrong interface.

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_DHCPS_Start()`.

Example

```
IP_DHCPS_ConfigGWAddr(0, IP_BYTES2ADDR(192, 168, 12, 1));
IP_DHCPS_ConfigPool(0, IP_BYTES2ADDR(192, 168, 12, 11), 0xFFFF0000, 20);
IP_DHCPS_Init(0);
IP_DHCPS_Start(0);
```


10.2.3 IP_DHCPS_ConfigMaxLeaseTime()

Description

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

Prototype

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

Parameters

Parameter	Description
IFIndex	Zero-based interface index.
Seconds	Maximum lease time in seconds. Default 7200s => 2h. Up to 4294967 seconds, converted into ms this is the maximum we can store in an U32. 0xFFFFFFFF to grant infinite if asked for.

Return value

0	O.K.
IP_ERR_MISC	Error, server already started.
IP_ERR_PARAM	Error, wrong interface or value for lease time invalid.

Additional information

Optional. Needs to be called before activating the DHCP server for this interface with IP_DHCPS_Start().

Example

```
IP_DHCPS_ConfigMaxLeaseTime(0, 7200);
IP_DHCPS_ConfigPool(0, IP_BYTES2ADDR(192, 168, 12, 11), 0xFFFF0000, 20);
IP_DHCPS_Init(0);
IP_DHCPS_Start(0);
```


10.2.4 IP_DHCPS_ConfigPool()

Description

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

Prototype

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

Parameters

Parameter	Description
IFIndex	Zero-based interface index of the server to configure.
StartIPAddr	First IP address of the pool in host order.
SNMask	Subnet mask in host order.
PoolSize	Number of addresses in the pool starting from StartIPAddr. The pool size has to be at least 1.

Return value

0	O.K.
IP_ERR_MISC	Error, server already started.
IP_ERR_PARAM	Error, wrong interface.

Additional information

Optional. Needs to be called before activating the DHCP server for this interface with IP_DHCPS_Start().

Example

```
IP_DHCPS_ConfigPool(0, IP_BYTES2ADDR(192, 168, 12, 11), 0xFFFF0000, 20);
IP_DHCPS_Init(0);
IP_DHCPS_Start(0);
```


10.2.5 IP_DHCPS_Halt()

Description

Stops DHCP server activity for the passed interface.

Prototype

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

Parameters

Parameter	Description
IFIndex	Zero-based interface index.

10.2.6 IP_DHCPS_Init()

Description

Initializes the DHCP server for the specified interface.

Prototype

```
int IP_DHCPS_Init(unsigned IFIndex);
```

Parameters

Parameter	Description
IFIndex	Zero-based interface index.

Return value

0	O.K.
IP_ERR_MISC	Error, server already initialized.
IP_ERR_NOMEM	Error, not enough memory.
IP_ERR_PARAM	Error, wrong interface.

Additional information

This function is obsolete. Its functionality has been implemented into IP_DHCPS_ConfigPool() as this needs to be called anyhow. This function is a dummy for the moment, so it does not hurt to call it like before.

10.2.7 IP_DHCPS_SetReservedAddresses()

Description

Sets a configuration for IP addresses to reserve for specific MAC addresses or HostNames (or both).

Prototype

```
int IP_DHCPS_SetReservedAddresses(      unsigned                 IFIndex,
                                  const IP_DHCPS_RESERVE_ADDR * paAddr,
                                        unsigned                NumAddr);
```

Parameters

Parameter	Description
IFIndex	Zero-based interface index.
paAddr	Pointer to array of IP_DHCPS_RESERVE_ADDR addresses.
NumAddr	Number of addresses at paAddr .

Return value

< 0 Error
= 0 O.K.

Additional information

For the moment the global configuration for subnet mask, gateway and DNS for the server on this interface is used.

IP addresses to be reserved are not limited to addresses of the configured pool. Of course addresses need to be within the configured subnet to work as expected.

Example

```
const IP_DHCPS_RESERVE_ADDR _aReserved[] = {
// HW addr.                              , IP addr.,                    , HostName
  {(const U8*)"\x00\x0C\x29\x76\xE7\x0B", IP_BYTES2ADDR(192,168,12,20), NULL},      // Reserve by HW addr. only.
  {(const U8*)"\x00\x22\xC7\xAF\xFC\x25", IP_BYTES2ADDR(192,168,12,16), "oliver"},  // Reserve by HW addr. AND Hostname
 (both have to match).
  {NULL                                , IP_BYTES2ADDR(192,168,12,17), "sven"},    // Reserve by Hostname first
 (or only).
  {(const U8*)"\x00\x22\xC7\xAF\xFC\x30", IP_BYTES2ADDR(192,168,12,17), NULL},      // Reserve by HW addr. second.
  {(const U8*)"\xB8\x27\xEB\xC7\x96\x5F", IP_BYTES2ADDR(192,168,20,55), NULL},      // Reserve by HW addr. only.
};

static void _StartServer(void) {
  IP_DHCPS_ConfigPool(0, IP_BYTES2ADDR(192, 168, 12, 11), IP_BYTES2ADDR(255, 255, 0, 0), 20);
  IP_DHCPS_Init(0);
  IP_DHCPS_SetReservedAddresses(0, _aReserved, SEGGER_COUNTOF(_aReserved));
  IP_DHCPS_Start(0);
}
```


10.2.8 IP_DHCPS_SetVendorOptionsCallback()

Description

This function sets a callback that is executed when sending response to a client and the client has sent a vendor class identifier (DHCP option 60). It can be used to add vendor specific options to a DHCP response to a client.

Prototype

```
void IP_DHCPS_SetVendorOptionsCallback(IP_DHCPS_GET_VENDOR_OPTION_FUNC * pf);
```

Parameters

Parameter	Description
pf	Callback of type IP_DHCPS_GET_VENDOR_OPTION_FUNC that is asked for vendor specific options.

Example

```
/*********************************************************************
*
*       _cbDHCPs_AddVendorOptions()
*
*  Function description
*    Adds DHCP vendor specific options to our server replies.
*
*  Parameters
*    IFaceId : Zero-based interface index.
*    pInfo   : Further information about the vendor of the client.
*    ppOption: Pointer to the pointer where to add further options.
*              The dereferenced pointer needs to be incremented
*              by the number of bytes added. Type and length bytes
*              need to be added by the callback as well.
*    NumBytes: Number of free bytes that can be used to store
*              options from the callback.
*/
static void _cbDHCPs_AddVendorOptions(unsigned IFaceId, IP_DHCPS_VENDOR_OPTION_INFO* pInfo, U8** ppOption, unsigned NumBytes) {
  U8*      pVendorClassId;
  U8*      pOption;
  unsigned VendorClassIdLen;

  IP_USE_PARA(IFaceId);

  pOption = *ppOption;  // Get the location where to add our options aka borrow the pointer.
  //
  // Parse the vendor class id.
  //
  pVendorClassId   = pInfo->pVendorClassId;        // Points to the type which should always be DHCP option 60.
  pVendorClassId++;                                // proceed to the length field.
  VendorClassIdLen = (unsigned)*pVendorClassId++;  // Get the length byte and proceed to the actual non-terminated vendor string.
  //
  // Check if the vendor class identifier is known to us.
  //
  if ((IP_MEMCMP(pVendorClassId, "MSFT 5.0", VendorClassIdLen) == 0) &&
     (NumBytes >= 8u)) {  // Also check if we have enough space to add the option.
    //
    // Identified a Microsoft device that supports vendor-specific options.
    // More information about this can be found at the following location:
    //   * https://msdn.microsoft.com/en-us/library/cc227279.aspx
    //
    // Information about the vendor-specific options supported for Microsoft
    // devices can be found here:
    //   * [1] https://msdn.microsoft.com/en-us/library/cc227275.aspx
    //   * [2] https://msdn.microsoft.com/en-us/library/cc227276.aspx
    //
    // A common task is to disable NetBIOS (over TCP/IP) via DHCP
    // if your clients primarily use other techniques and you want
    // to speed up discovery of them by name. Typically one method
    // will be tested after each other which means that each method
    // used costs additional time before your desired discovery
    // method finally might be used.
    //
    *pOption++  =  43u;                   // Add an option field of type 43 "Vendor-Specific Information".
    *pOption++  =   6u;                   // Add length field with value 6 for the actual 6 bytes vendor-specific content.
    *pOption++  = 0x01;                   // [1] "Microsoft Disable NetBIOS Option (section 2.2.2.1)"
    *pOption++  = 0x04;                   // [2] "Vendor-specific Option Length"
    IP_StoreU32BE(pOption, 0x00000002uL);  // [2] "Vendor-specific Option Data" "Disables NetBIOS over TCP/IP for that network interface."
    pOption    += 4;
  }
  *ppOption = pOption;  // Write back the borrowed pointer so the DHCP server internal code knows where to continue.
}

void main(void) {
  ...
  IP_DHCPS_SetVendorOptionsCallback(_cbDHCPs_AddVendorOptions);
  ...
}
```


10.2.9 IP_DHCPS_Start()

Description

Starts the DHCP server for the specified interface.

Prototype

```
int IP_DHCPS_Start(unsigned IFIndex);
```

Parameters

Parameter	Description
IFIndex	Zero-based interface index.

Return value

0	O.K.
IP_ERR_MISC	Error, server already started or not initialized/configured.
IP_ERR_NOMEM	Error, not enough memory.
IP_ERR_PARAM	Error, wrong interface.

Additional information

IP_DHCPS_Init() and IP_DHCPS_ConfigPool() needs to be called before activating the DHCP server for an interface in order to set at least the minimum configurations.

10.3 Data structures

10.3.1 IP_DHCPS_RESERVE_ADDR

Description

Reserves a DHCP IPv4 address via HW address, hostname or both.

Type definition

```
typedef struct {
  const U8   * pHWAddr;
  U32          IPAddr;
  const char * sHostName;
} IP_DHCPS_RESERVE_ADDR;
```

Structure members

Member	Description
pHWAddr	Client HW/MAC address to reserve to. Can be NULL.
IPAddr	IPv4 address to reserve in host endianness. Does not need to be from the DHCP pool itself.
sHostName	Client hostname to reserve to. Can be NULL.

10.3.2 IP_DHCPS_GET_VENDOR_OPTION_INFO

Description

Returns information about the vendor specific identifier received with DHCP option 60.

Type definition

```
typedef struct {
  U8 * pVendorClassId;
} IP_DHCPS_GET_VENDOR_OPTION_INFO;
```

Structure members

Member	Description
pVendorClassId	Pointer to the DHCP option 60 field received from a client including type and length bytes. A typical example would be Type: 60, Len: 8 and Value: 'M' 'S' 'F' 'T' ' ' '5' '.' '0' for a Microsoft client that supports vendor specific DHCP option 43 commands.

10.3.3 IP_DHCPS_GET_VENDOR_OPTION_FUNC

Description

Inserts a vendor specific configuration for DHCP option 43.

Type definition

```
typedef void (IP_DHCPS_GET_VENDOR_OPTION_FUNC)
                                      (unsigned                          IFaceId,
                                       IP_DHCPS_GET_VENDOR_OPTION_INFO * pInfo,
                                       U8                             ** ppOption,
                                       unsigned                          NumBytes);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pInfo	Further information of type IP_DHCPS_GET_VENDOR_OPTION_INFO about the vendor of the client.
ppOption	Pointer to the pointer where to add further options. The dereferenced pointer needs to be incremented by the number of bytes added. Type and length bytes need to be added by the callback as well.
NumBytes	Number of free bytes that can be used to store options from the callback.

10.4 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.

10.4.1 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
emNet DHCP server	approximately 2.0 kByte

10.4.2 ROM usage on a Cortex-M3 system

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

Addon	ROM
emNet DHCP server	approximately 2.0 kByte

10.4.3 RAM usage

Addon	RAM
emNet DHCP server	approximately 200 bytes

Chapter 11

mDNS Server (Add-on)

The emNet implementation of mDNS server which stands for multicast DNS server is an optional extension to emNet. It makes your target easily discoverable and advertising services available throughout your network.

For the target IP address identification, this add-on also replies to Microsoft LLMNR requests.

11.1 emNet mDNS

The emNet mDNS implementation is an optional extension which can be seamlessly integrated into your TCP/IP application. It allows your target to be easily identified with a small memory footprint.

The mDNS server module implements the relevant parts of the following RFCs.

Document	Download
Multicast DNS	Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc6762.txt*
Link-Local Multicast Name Resolution (LLMNR)	Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc4795.txt*
DNS-Based Service Discovery	Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc6763.txt*
A DNS RR for specifying the location of services (DNS SRV)	Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2782.txt*

The following table shows the contents of the emNet root directory:

Directory	Content
`Application`	Contains an example application that run's a simple mDNS server example.
`IP`	Contains the mDNS server file, `IP_DISCOVER.c`.

11.2 Feature list

- Low memory footprint.
- Makes your target easily discoverable.
- Easy to implement.

11.3 Requirements

TCP/IP stack

The emNet mDNS server implementation requires the emNet TCP/IP stack.

11.4 Multicast DNS background

Multicast DNS allows to find devices in an environment without the support of an actual DNS server. A DNS request is sent to a specific multicast address on a specific port. Servers are listening on this multicast address and handling the requests.

Multicast DNS handles only local systems and doen’t replace a real DNS for request outside the local network.

11.4.1 Hostname resolution

In order to get the IP address of a target by its name, two records could be sent:

- A: To get the IPv4 address.
- AAAA: To get the IPv6 address.

Apple and Microsoft are both proposing a similar solution but using different multicast IP addresses and ports. The Add-on is handling both specifications for A and AAAA requests.

The hostname is set in the configuration structure:

```
static const IP_DNS_SERVER_CONFIG     _Config = {
  .sHostname  = "mytarget.local",
  .TTL        = 120,
  .NumConfig  = 3,          // Could be 0 for name resolution only
  .apSDConfig = _SDConfig,  // DNS-SD config, could be NULL.
};
```


11.4.2 Service discovery (mDNS-SD)

The add-on also provides the definition of some services through additional records:

- PTR: Pointer record.
- SRV: Service record.
- TXT: Text record.

The service discovery is only available through the Apple multicast address (use of Bonjour), or equivalent on linux machines (like avahi).

For example, if a target runs a web server, a possible configuration is:

```
static const IP_DNS_SERVER_SD_CONFIG   _SDConfig[] = {
  {
    .Type                       = IP_DNS_SERVER_TYPE_PTR,     ❶
    .Flags                      = IP_DNS_SERVER_FLAG_FLUSH,
    .TTL                        = 0,
    .Config = {
      .PTR   = {
                 .sName         = "_http._tcp.local",
                 .sDomainName   = "myserver._http._tcp.local"
               }
             }
  },
  {
    .Type                       = IP_DNS_SERVER_TYPE_SRV,     ❷
    .Flags                      = IP_DNS_SERVER_FLAG_FLUSH,
    .TTL                        = 0,
    .Config = {
      .SRV   = {
                 .sName         = "myserver._http._tcp.local",
                 .Priority      = 0,
                 .Weight        = 0,
                 .Port          = 80,
                 .sTarget       = "mytarget.local"
               }
             }
  },
  {
    .Type                       = IP_DNS_SERVER_TYPE_TXT,     ❸
    .Flags                      = IP_DNS_SERVER_FLAG_FLUSH,
    .TTL                        = 0,
    .Config = {
      .TXT   = {
                 .sName         = "myserver._http._tcp.local",
                 .sTXT          = "PATH=/"
               }
             }
  }
};
```

❶ PTR record

: The PTR record indicates that an HTTP server runs at "`myserver._http._tcp.local`"

❷ SRV record

: The SRV record gives indication on the port number (80) and the actual local target name (mytarget)

❸ TXT record

: The TXT record gives additional information, for example the path to the web server.

It is possible to add A and AAAA records, but they are not needed if the target name corresponds to the target host name.

11.5 API functions

Function	Description
IP_MDNS_SERVER_Start()	Starts the LLMNR/mDNS DNS-SD discovery service.
IP_MDNS_SERVER_Stop()	Stops the LLMNR/mDNS DNS-SD discovery service.

11.5.1 IP_MDNS_SERVER_Start()

Description

Starts the LLMNR/mDNS DNS-SD discovery service.

Prototype

```
int IP_MDNS_SERVER_Start(const IP_DNS_SERVER_CONFIG * pConfig);
```

Parameters

Parameter	Description
pConfig	Pointer to the configuration array.

Return value

= 0 O.K.
< 0 Error

Example

Configuration should define local names.

```
static const IP_DNS_SERVER_CONFIG _Config = {
  .sHostname  = "mytarget.local",
  .TTL        = 120,
  .NumConfig  = 0,     // No DNS-SD configuration.
  .apSDConfig = NULL
};
IP_MDNS_SERVER_Start(&_Config);
```


11.5.2 IP_MDNS_SERVER_Stop()

Description

Stops the LLMNR/mDNS DNS-SD discovery service.

Prototype

```
int IP_MDNS_SERVER_Stop(void);
```

Return value

0	OK.

11.6 Data structures

11.6.1 Structure IP_DNS_SERVER_CONFIG

Description

This is the main configuration of the mDNS server.

Prototype

```
typedef struct {
  const char*                     sHostname;
        U32                       TTL;
        unsigned                  NumConfig;
  const IP_DNS_SERVER_SD_CONFIG*  apSDConfig;
} IP_DNS_SERVER_CONFIG;
```

Member	Description
sHostname	Pointer on a null terminated string corresponding to the host name (for example “mytarget.local”)
TTL	Time to live in seconds. If set to 0 a default value defined in DNS_TTL_INIT is used.
NumConfig	Number of mDNS-SD configuration pointed by apSDConfig. Could be 0.
apSDConfig	Array of mDNS-SD configuration. Could be NULL if NumConfig is 0.

11.6.2 Structure IP_DNS_SERVER_SD_CONFIG

Description

Configuration of a mDNS-SD entry.

Prototype

```
typedef struct {
  U32                         TTL;
  union {
    IP_DNS_SERVER_A           A;
#if IP_SUPPORT_IPV6
    IP_DNS_SERVER_AAAA        AAAA;
#endif
    IP_DNS_SERVER_PTR         PTR;
    IP_DNS_SERVER_SRV         SRV;
    IP_DNS_SERVER_TXT         TXT;
  } Config;
  U8                          Type;
  U8                          Flags;
} IP_DNS_SERVER_SD_CONFIG;
```

Member	Description
TTL	Time to live in seconds for this entry. If set to 0 the main TTL value from the structure IP_DNS_SERVER_CONFIG is used.
A	A record description. Not needed for the hostname. See IP_DNS_SERVER_A .
AAAA	AAA record description. Not needed for the hostname. See IP_DNS_SERVER_AAAA .
PTR	Pointer record description. See IP_DNS_SERVER_PTR .
SRV	Service record description. See IP_DNS_SERVER_SRV .
TXT	Text record description. See IP_DNS_SERVER_TXT .
Type	This is the type of the entry: - IP_DNS_SERVER_TYPE_A - IP_DNS_SERVER_TYPE_PTR - IP_DNS_SERVER_TYPE_TXT - IP_DNS_SERVER_TYPE_SRV - IP_DNS_SERVER_TYPE_AAAA
Flags	Optional configuration flags for this entry.

Flags	Description
IP_DNS_SERVER_FLAG_FLUSH	Sets the FLUSH bit when sending a response that contains this entry. The FLUSH bit should be set on all unique resources like the primary host name or in general A and AAAA records. Unique entries in repsonses are meant to FLUSH all previously returned configurations. We do not set this flag automatically for ANY entry (not even A and AAAA) to allow maximum freedom in your configuration.

11.6.3 Structure IP_DNS_SERVER_A

Description

Description of a A record entry (IPv4 IP address). This is not needed to have an entry for the host name. An 'A' request with the host name gets automatically a reply with the current IP address of the interface on which the request is received.

If the field `IPAddr` is set to 0, the IP address of the host will be used automatically.

Prototype

```
typedef struct {
  char*                   sName;
  IP_ADDR                 IPAddr;
} IP_DNS_SERVER_A;
```

Member	Description
`sName`	Null terminated string of the server name.
`IPAddr`	IPv4 address of the server name.

11.6.4 Structure IP_DNS_SERVER_AAAA

Description

Description of a AAAA record entry (IPv6 IP address). This is not needed to have an entry for the host name. An 'AAAA' request with the host name gets automatically a reply with the current IP address of the interface on which the request is received.

If the field `aIPAddrV6` is completely set to 0 (the 16 bytes are all 0), the IP address of the host will be used automatically.

Prototype

```
typedef struct {
  char*                   sName;
  U8                      aIPAddrV6[IPV6_ADDR_LEN];
} IP_DNS_SERVER_AAAA;
```

Member	Description
`sName`	Null terminated string of the server name.
`aIPAddrV6`	IPv6 address of the server name.

11.6.5 Structure IP_DNS_SERVER_PTR

Description

Description of a PTR record entry. This could either convert an IP address into a server name, for example 1.0.168.192.in-addr.arpa into myserver.local. Or this could be used to indicate the server that provides a service (like `_http._tcp.local`).

Prototype

```
typedef struct {
  char*                    sName;
  char*                    sDomainName;
} IP_DNS_SERVER_PTR;
```

Member	Description
sName	Null terminating string defining the entry that is requested. (what appears in the request).
sDomainName	Null terminating string. This is the reply. If set to `NULL`, the sHostname of the main config `IP_DNS_SERVER_CONFIG` is used.

11.6.6 Structure IP_DNS_SERVER_SRV

Description

Description of a SRV record entry. This describes which server provides a service and additional information like priority, port, …

Prototype

```
typedef struct {
  char*                   sName;
  U16                     Priority;
  U16                     Weight;
  U16                     Port;
  char*                   sTarget;      // If NULL, hostname will be used.
} IP_DNS_SERVER_SRV;
```

Member	Description
sName	Null terminating string defining the entry that is requested, Service, Protocol and Name are concatenated. (what appears in the request).
Priority	Priority value: 0 is the heigher priority.
Weight	Weight to balance between equivalent servers with the same priority.
Port	Port providing the service.
sTarget	Null terminating string. This is the server name. If set to NULL, the sHostname of the main config IP_DNS_SERVER_CONFIG is used.

11.6.7 Structure IP_DNS_SERVER_TXT

Description

Description of a TXT record entry. This describes some textual parameters. There could be many TXT records for the same name defining many parameters, but in this case, they should be placed next to one another in the configuration structure.

Prototype

```
typedef struct {
  char*                    sName;
  char*                    sTXT;
} IP_DNS_SERVER_TXT;
```

Member	Description
sName	Null terminating string defining the entry that is requested. (what appears in the request).
sTXT	Null terminating string defining one text entry. For example “Version=1”

11.7 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the CoAP client/server presented in the tables below have been measured on a Cortex-M4 system with the default configuration.

11.7.1 ROM usage on a Cortex-M4 system

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

Addon	ROM
emNet mDNS server	approximately 3.1 kBytes

11.7.2 RAM usage

The add-on uses a small internal table for the multicast UDP management.

Addon	RAM
emNet mDNS server	approximately 0.2 kBytes

Chapter 12

DNS Server (Add-on)

This add-on provides a simple DNS server which allows for a server to handle the DNS requests it receives. This could be used to gives the IP address of the target as a reply to a server enquiry.

It is ideally coupled with the *DHCP server (Add-on)* on page 436.

12.1 emNet DNS server

The emNet DNS implementation is an optional extension which can be seamlessly integrated into your TCP/IP application.

The DNS server module implements the relevant parts of the following RFCs.

Document	Download
DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION	Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1035.txt*
DNS-Based Service Discovery	Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc6763.txt*
A DNS RR for specifying the location of services (DNS SRV)	Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2782.txt*

The following table shows the contents of the emNet root directory:

Directory	Content
`IP`	Contains the mDNS server file, `IP_DISCOVER.c`.

12.2 Feature list

- Low memory footprint.
- Makes your target easily discoverable.
- Easy to implement.

12.3 Requirements

TCP/IP stack

The emNet DNS server implementation requires the emNet TCP/IP stack.

12.4 Implementation

The emNet simple DNS server used the same mechanism and configuration as the *mDNS Server (Add-on)* on page 452. Thus the structures and parameters won't be described further in this chapter.

The only difference is that target name definition are not local anymore since a DNS is faked. Thus the ".local" extension is not needed anymore.

12.5 API functions

Function	Description
IP_DNS_SERVER_Start()	Starts the simple DNS service.
IP_DNS_SERVER_Stop()	Stops the simple DNS service.

12.5.1 IP_DNS_SERVER_Start()

Description

Starts the simple DNS service.

Prototype

```
int IP_DNS_SERVER_Start(const IP_DNS_SERVER_CONFIG * pConfig);
```

Parameters

Parameter	Description
pConfig	Pointer to the fake DNS configuration.

Return value

0	OK.
-1	Error. Could not open connection(s) or DNS service not supported (IP_SUPPORT_FAKE_DNS = 0).

Example

Configuration should define local names.

```
static const IP_DNS_SERVER_CONFIG _Config = {
  .sHostname  = "mytarget.eth",
  .TTL        = 120,
  .NumConfig  = 0,     // No DNS-SD configuration.
  .apSDConfig = NULL
};
IP_DNS_SERVER_Start(&_Config);
```


12.5.2 IP_DNS_SERVER_Stop()

Description

Stops the simple DNS service.

Prototype

```
int IP_DNS_SERVER_Stop(void);
```

Return value

= 0	O.K.
< 0	Error

12.6 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the CoAP client/server presented in the tables below have been measured on a Cortex-M4 system with the default configuration.

In addition to the existing mDNS server add-on, the DNS server add-on adds approximately 0.2 kBytes of ROM.

Chapter 13

AutoIP

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

13.1 emNet AutoIP backgrounds

The emNet AutoIP module adds the dynamic configuration of IPv4 Link-Local addresses to emNet. 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.

13.2 API functions

Function	Description
IP_AutoIP_Activate()	Activates AutoIP negotiation for the specified interface.
IP_AutoIP_Halt()	Stops AutoIP activity for the passed interface.
IP_AutoIP_SetUserCallback()	This function allows the caller to set a callback for an interface.
IP_AutoIP_SetStartIP()	Sets the IP address which will be used for the first configuration try.

13.2.1 IP_AutoIP_Activate()

Description

Activates AutoIP negotiation for the specified interface.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero based interface index.

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 the stack will generate an IP address similar to 169.254.xxx.xxx. The subnet mask of is always 255.255.0.0.

In emNet 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 1220 and *IP_AddLogFilter* on page 1218 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.
```


13.2.2 IP_AutoIP_Halt()

Description

Stops AutoIP activity for the passed interface.

Prototype

```
int IP_AutoIP_Halt(unsigned IFaceId,
                   char     KeepIP);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
KeepIP	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.

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.

13.2.3 IP_AutoIP_SetUserCallback()

Description

This function allows the caller to set a callback for an interface. The callback is called with every status change.

Prototype

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

Parameters

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

Additional information

This mechanism 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);
```


13.2.4 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);
```

Parameters

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

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 startingpoint of the AutoIP functionality.

13.3 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.

13.3.1 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
emNet AutoIP module	approximately 1.1 kByte

13.3.2 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
emNet AutoIP module	approximately 1.0 kByte

13.3.3 RAM usage

Addon	RAM
emNet AutoIP module	approximately 0.7 kByte

Chapter 14

Address Collision Detection (ACD)

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

14.1 emNet ACD module

The emNet ACD module allows to detect and react to IPv4 address collisions on the network. The typical case is that one or more hosts on the network use the same IPv4 address. To detect other hosts using the same IP address, ACD can use passive listening for ARP packets sent by hosts as well as active probing for the IP address.

The ACD module implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 5227]	IPv4 Address Conflict Detection Direct download: *https://datatracker.ietf.org/doc/html/rfc5227*

14.2 API functions

Function	Description
IP_ACD_Activate()	Activates the address conflict detection (ACD) for the specified interface.
IP_ACD_ActivateEx()	Activates the address conflict detection (ACD) for the specified interface and allows extended configuration.
IP_ACD_Config()	Configures the address conflict detection (ACD) behavior for startup and in case of conflicts.
IP_ACD_EndAnnounce()	Ends sending further announce messages when in IP_ACD_STATE_ANNOUNCE_SEND_GARP state.
IP_ACD_Halt()	De-Activates the address conflict detection (ACD) for the specified interface.
IP_ACD_UpdateBackgroundPeriod()	Updates the “BackgroundPeriod” when in IP_ACD_STATE_ACTIVE_* state.

14.2.1 IP_ACD_Activate()

Description

Activates the address conflict detection (ACD) for the specified interface.

Prototype

```
int IP_ACD_Activate(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

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.
< 0 Error, no memory.

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.

It is the responsibility of the application to make sure that ACTIVATE is only called when the interface is UP. As ACD only makes sense for an interface in state UP, the ACTIVATE call might actively wait for the interface state to change.

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

In emNet 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 1220 and *`IP_AddLogFilter`* on page 1218 for further information about the enabling terminal I/O.

14.2.2 IP_ACD_ActivateEx()

Description

Activates the address conflict detection (ACD) for the specified interface and allows extended configuration.

Prototype

```
int IP_ACD_ActivateEx(      unsigned               IFaceId,
                            IP_ACD_ON_INFO_FUNC  * pfOnInfo,
                      const IP_ACD_EX_CONFIG     * pConfig,
                            unsigned               NonBlocking);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pfOnInfo	Callback of type IP_ACD_ON_INFO_FUNC to be notified about state changes and events.
pConfig	Pointer to configuration of type IP_ACD_EX_CONFIG .
NonBlocking	• 0: Call is blocking and waits for the operations to finish. • 1: Call is non-blocking and returns instantly.

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.
< 0 Error, no memory.

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.

It is the responsibility of the application to make sure that ACTIVATE is only called when the interface is UP. As ACD only makes sense for an interface in state UP, the ACTIVATE call might actively wait for the interface state to change.

14.2.3 IP_ACD_Config()

Description

Configures the address conflict detection (ACD) behavior for startup and in case of conflicts.

Prototype

```
int IP_ACD_Config(      unsigned   IFaceId,
                        unsigned   NumProbes,
                        unsigned   DefendInterval,
                  const ACD_FUNC * pAPI);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
NumProbes	Number of ARP probes to send upon activating ACD before declaring the actual used IP address to be free to be used.
DefendInterval	Interval [ms] in which the currently active IP address is being known as defended after taking action.
pAPI	Pointer to callback table of type ACD_FUNC .

Return value

= 0 O.K.
< 0 Error, no memory.

14.2.4 IP_ACD_EndAnnounce()

Description

Ends sending further announce messages when in IP_ACD_STATE_ANNOUNCE_SEND_GARP state.

Prototype

```
void IP_ACD_EndAnnounce(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Additional information

This routine is designed to be called either from the ACD information callback or from another place in the application to end an ongoing sending of announce messages early when in the IP_ACD_STATE_ANNOUNCE_SEND_GARP state. Ending the IP_ACD_STATE_ANNOUNCE_SEND_GARP state early might be necessary for example when implementing EtherNet/IP "QuickConnect" and communication is established while still sending announce messages.

At the moment using this routine is limited to the IP_ACD_STATE_ANNOUNCE_SEND_GARP state and is internally checked. If important to your application you should ensure that this is the case in your application as this internal check might be subject to change in the future.

Calling this routine sends the state machine into ACTIVE or PASSIVE mode. In ACTIVE mode it is possible to influence the time before sending the first probe by overriding the proposed delay in the info callback for the IP_ACD_STATE_ACTIVE_WAIT_BEFORE_BG_PROBES state.

14.2.5 IP_ACD_Halt()

Description

De-Activates the address conflict detection (ACD) for the specified interface.

Prototype

```
void IP_ACD_Halt(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

14.2.6 IP_ACD_UpdateBackgroundPeriod()

Description

Updates the "BackgroundPeriod" when in IP_ACD_STATE_ACTIVE_* state.

Prototype

```
void IP_ACD_UpdateBackgroundPeriod(unsigned IFaceId,
                                   unsigned BackgroundPeriod);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
BackgroundPeriod	Background period in milliseconds.

Additional information

This routine is designed to be called either from the ACD information callback or from another place in the application to modify the period used when sending background probes in ACTIVE mode in the IP_ACD_STATE_ACTIVE_* states.

At the moment using this routine is limited to the IP_ACD_STATE_ACTIVE_* states and is internally checked. If important to your application you should ensure that this is the case in your application as this internal check might be subject to change in the future.

Calling this routine sends the state machine into ACTIVE mode and ends states such as IP_ACD_STATE_ACTIVE_WAIT_BEFORE_BG_PROBES if the state machine is currently in this state.

Changing the background probing period can be used to enter or leave the EtherNet/IP "SemiActiveProbe" mode.

14.3 Data structures

14.3.1 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 {
  U32 (*pfRenewIPAddr)(unsigned IFace);
  int (*pfDefend)     (unsigned IFace);
  int (*pfRestart)    (unsigned IFace);
} 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.

14.3.2 IP_ACD_EX_CONFIG

Description

Used to configure the extended ACD functionality.

Type definition

```
typedef struct {
  U32           IPAddr;
  unsigned      BackgroundPeriod;
  unsigned      NumProbes;
  unsigned      DefendInterval;
  unsigned      NumAnnouncements;
  unsigned      AnnounceInterval;
  U8            AssignAddressManually;
  IP_ACD_STATE  InitState;
} IP_ACD_EX_CONFIG;
```

Structure members

Member	Description
IPAddr	IPv4 start address to use in host endianness.
BackgroundPeriod	Period [ms] in which ACD will send probes running in the background.
NumProbes	Number of ARP probes to send upon activating ACD before declaring the actual used IP address to be free to be used. 0 to use default.
DefendInterval	Interval [ms] in which the currently active IP address is being known as defended after taking action. 0 to use default.
NumAnnouncements	Number of announcements to send when using a free address. The address can already be used at this point. 0 to use default.
AnnounceInterval	Time [ms] between announcements to send. 0 to use default.
AssignAddressManually	Configures if probed address is assigned automatically to the interface if free. • 0: Off (default), address is automatically to the interface, using the existing subnet mask. • 1: On, address is only reported via the IP_ACD_INFO.IPAddr member in the IP_ACD_STATE_INIT_WAIT_BEFORE_ANNOUNCE state. The user needs to manually assign it to the interface along with the desired subnet mask. Assigning an address manually might affect ACD effectiveness on virtual interfaces such as being used for multiple addresses on one single physical interface. ARP/ACD might not be able to correctly select the virtual interface for some operations until the address has finally been assigned to the interface.
InitState	Initial state for the ACD state machine upon activating ACD. The following states are supported: • IP_ACD_STATE_DISABLED Default behavior starting ACD listening for potential conflicts at the beginning. • IP_ACD_STATE_ANNOUNCE_SEND_GARP Skips the initial listening phase and starts by directly sending the first the first announcement. This can be used to implement EtherNet/IP “QuickConnect” behavior.

14.3.3 IP_ACD_ANNOUNCE_INFO

Description

Returns information about the latest ACD announce about using a free and previously probed address.

Type definition

```
typedef struct {
  unsigned  AnnouncementsLeft;
} IP_ACD_ANNOUNCE_INFO;
```

Structure members

Member	Description
AnnouncementsLeft	Number of announements left to send.

14.3.4 IP_ACD_COLLISION_INFO

Description

Returns information about the latest ACD collision.

Type definition

```
typedef struct {
  IP_PACKET * pPacket;
  U32         DefendTimeout;
  unsigned    ProbesLeft;
} IP_ACD_COLLISION_INFO;
```

Structure members

Member	Description
pPacket	Pointer to the packet that caused the collision (pPacket->pData points to the ARP header).
DefendTimeout	System timestamp of when the defend window ends.
ProbesLeft	Number of INIT probes left to send.

14.3.5 IP_ACD_WAIT_INFO

Description

Returns information about a delay/wait before the next step. This can be a delay before sending the very first probe for INIT or a delay between each probe sent during the INIT phase.

Type definition

```
typedef struct {
  unsigned           WaitMin;
  unsigned volatile  WaitTime;
  unsigned           WaitMax;
} IP_ACD_WAIT_INFO;
```

Structure members

Member	Description
WaitMin	Suggested minimum wait time [ms].
WaitTime	Wait time before the next state that is used (does not have to obey min./max. suggestion). This value can be overwritten and is evaluated after returning from the callback.
WaitMax	Suggested maximum wait time [ms]

Additional information

The stack makes suggestions using the structure members as well as presenting the actual value that will be used in the WaitTime member. You can overwrite the WaitTime member as it is then evaluated after returning from the callback and its new value value is then used.

14.3.6 IP_ACD_INFO

Description

Returns information about the current ACD status. The ACD info callback parameter `State` has to be evaluated for further information if there is more info about the new state and what part of the union is the information to look at.

Type definition

```
typedef struct {
  U32                          IPAddr;
  IP_ACD_STATE                 State;
  IP_ACD_STATE                 OldState;
  IP_ACD_LOSE_DEFEND_ADDRESS   Defend;
  IP_ACD_KEEP_DISCARD_PACKET   DiscardPacket;
} IP_ACD_INFO;
```

Structure members

Member	Description
`IPAddr`	IPv4 address (in host endianness) that gets assigned to the interface or would be assigned to the interface if `IP_ACD_EX_CONFIG.AssignAddressManually` is NOT used. Currently only valid with the `IP_ACD_STATE_INIT_WAIT_BEFORE_ANNOUNCE` state.
`State`	Type of information and what part of the union to look at. The `State` member is followed by a union that might not be correctly displayed or completely missing in the manual. The following information describes this union part: • `IP_ACD_STATE_EVENT_COLLISION`: Information about the latest ACD collision can be found in `pInfo->`Data.Collision in form of `IP_ACD_COLLISION_INFO` .
`OldState`	Previous state. Might be the same as new state depending on actions executed in callbacks. If filtering is needed, this needs to be implemented in the application. When counting events like how many announcements have been sent, the `OldState` should be used for filtering as this reports the event handled immediately before or after reporting the state (change).
`Defend`	Suggestion from the stack whether to defend the IP address on a collision after INIT or not. This value is evaluated after returning from the callback. • `= IP_ACD_LOSE_ADDRESS` : Lose the address (typically if this is not the first conflict with a host and is within the defend window. • `= IP_ACD_DEFEND_ADDRESS`: `Defend` the address (anyhow).
`DiscardPacket`	Suggestion from the stack whether to keep or discard a packet contained in the state specific information structure (e.g. `IP_ACD_COLLISION_INFO`). This value is evaluated after returning from the callback. • `= IP_ACD_KEEP_PACKET` : Packet is kept and forwarded to the ARP module for further handling. • `= IP_ACD_DISCARD_PACKET`: Packet is discarded (e.g. to avoid ARP cache poisoning).

14.3.7 IP_ACD_ON_INFO_FUNC

Description

Callback executed whenever updated ACD information is available.

Type definition

```
typedef void (IP_ACD_ON_INFO_FUNC)(unsigned      IFaceId,
                                   IP_ACD_INFO * pInfo);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pInfo	Further information of type IP_ACD_INFO about the actual information available.

Additional information

Calling API like an ACTIVATE from the callback might produce another callback message. It is the responsibility of the application to avoid infinite recursion. Typically this is no problem as calling ACTIVATE again from the callback reporting the ACTIVATE state makes no sense.

14.4 EtherNet/IP usage

The standard ACD behavior and timers are typically sufficient to raise notifications in a regular network and detect configuration problems in non critical environments. Other environments and protocols used might be subject to more strict parameters to detect configuration problems faster and allow to react to them in a more prioritized way than just notifying about the potential problem on the network.

EtherNet/IP is such an environment/protocol that in its basic principles makes use of ACD as is but extends it by some specific behavior here and there. This section explains how to configure the ACD module for some of these EtherNet/IP specific requirements.

14.4.1 EtherNet/IP QuickConnect

QuickConnect capable devices power up in less than 300 ms and are able to establish a network connection in less than 200 ms. A QuickConnect is for example used when replacing tools in an EtherNet/IP network that need to be back to production basically instantaneous.

In QuickConnect mode ACD is not started in its slow probing initial state but shall be doing quick negotions with the network to decide if there is a collision or not. A typical configuration is to directly start announcing the IP that is to be used to the network in a quick paced manner to make sure this is received as early as possible by other hosts that might be subject to collision with the QuickConnect device entering the network.

The rapid announcing of its address is stopped early when detecting I/O communication with the device, confirming that based on network topology and switch ARP tables, communication ends up with this device after all.

EtherNet/IP QuickConnect example

QuickConnect configuration example sending 40 ARP announcements with a period of 25 ms before switching into active background probing with a period of 1 second.

```
static unsigned     _ACD_IFaceId = 0u;
static IP_ACD_STATE _ACD_State;

/*********************************************************************
*
*       ACD configuration
*/
static IP_ACD_EX_CONFIG _ACD_Config = {
  ...
  .BackgroundPeriod = 1000u,
  .NumAnnouncements = 40u,
  .AnnounceInterval = 25u,
  .InitState        = IP_ACD_STATE_ANNOUNCE_SEND_GARP,
  ...
};

/*********************************************************************
*
*       _cbOnInfo()
*
*  Function description
*    Callback executed whenever updated ACD information is available.
*
*  Parameters
*    IFaceId : Zero-based interface index.
*    pInfo   : Further information of type IP_ACD_INFO about the actual
*              information available.
*
*  Additional information
*    Calling API like an ACTIVATE from the callback might produce
*    another callback message. It is the responsibility of the application
*    to avoid infinite recursion. Typically this is no problem as calling
*    ACTIVATE again from the callback reporting the ACTIVATE state makes
*    no sense.
*/
static void _ACD_cbOnInfo(unsigned IFaceId, IP_ACD_INFO* pInfo) {
  IP_USE_PARA(IFaceId);

  _ACD_State = pInfo->State;
}

/*********************************************************************
*
*       _EIP_cbOnIO()
*
*  Function description
```



```
*    Callback executed upon EtherNet/IP I/O communication.
*
*  Additional information
*    This callback is executed when detecting I/O communication and
*    is used to end the QuickConnect rapid announcement phase early.
*/
static void _EIP_cbOnIO(void) {
  if (_ACD_State == IP_ACD_STATE_ANNOUNCE_SEND_GARP) {
    IP_ACD_EndAnnounce(_ACD_IFaceId);
  }
}

/*********************************************************************
*
*       MainTask
*/
void MainTask(void) {
  ...
  IP_Init();
  ...
  //
  // Wait for Link-UP.
  //
  ...
  IP_ACD_ActivateEx(_ACD_IFaceId, _ACD_cbOnInfo, &_ACD_Config, 0u);
  ...
}
```


14.4.2 EtherNet/IP SemiActiveProbe

If an interface is already successfully established and doing its background probing and another interface of this device reaches link-UP state, the device shall enter the so called `SemiActiveProbe` state. As the device has the same IP address for its multiple EtherNet/IP interfaces, it shall probe if it sees itself due to a wrong network topology.

A typical probing algorithm is to send two ARP probes with a period of 200 ms in the form of `DELAY => PROBE => DELAY => PROBE` . Once done, the device shall return to its regular ACD background probing cycle.

EtherNet/IP SemiActiveProbe example

SemiActiveProbe example sending two short probes with a period of 200 ms before returning back to its original background probing with a period of 1 second.

```
static unsigned               _ACD_IFaceId = 0u;
static unsigned               _ACD_SemiActiveProbesLeft;
static IP_HOOK_ON_LINK_CHANGE _ACD_LinkChangeHook;
static IP_ACD_STATE           _ACD_State;

/*********************************************************************
*
*       ACD configuration
*/
static IP_ACD_EX_CONFIG _ACD_Config = {
  ...
  .BackgroundPeriod = 1000u,
  ...
};

/*********************************************************************
*
*       _cbOnInfo()
*
*  Function description
*    Callback executed whenever updated ACD information is available.
*
*  Parameters
*    IFaceId : Zero-based interface index.
*    pInfo   : Further information of type IP_ACD_INFO about the actual
*              information available.
*
*  Additional information
*    Calling API like an ACTIVATE from the callback might produce
*    another callback message. It is the responsibility of the application
*    to avoid infinite recursion. Typically this is no problem as calling
*    ACTIVATE again from the callback reporting the ACTIVATE state makes
*    no sense.
*/
static void _ACD_cbOnInfo(unsigned IFaceId, IP_ACD_INFO* pInfo) {
  IP_USE_PARA(IFaceId);

  _ACD_State = pInfo->State;
  //
  // Handle SemiActiveProbe mode.
  // Use the OldState to avoid reacting to the transition into
  // the SemiActiveProbe mode. Reacting to the OldState means
  // to react to the end of the first 200 ms delay and sending
  // the first SemiActiveProbe .
  //
  if (pInfo->OldState == IP_ACD_STATE_ACTIVE_SEND_BG_PROBES) {
    //
    // Are we in SemiActiveProbe mode ?
    //
    if (_ACD_SemiActiveProbesLeft != 0u) {
      _ACD_SemiActiveProbesLeft--;
```



```
      //
      // Is this the last probe to send with SemiActiveProbe long period ?
      //
      if (_ACD_SemiActiveProbesLeft == 0u) {
        //
        // Return back to the regular background probing period.
        //
        IP_ACD_UpdateBackgroundPeriod(_ACD_IFaceId, 1000u);
      }
    }
  }
}

/*********************************************************************
*
*       _ACD_cbOnLinkChange()
*
*  Function description
*    Callback executed whenever the link state of an interface changes.
*
*  Parameters
*    IFaceId : Zero-based interface index.
*    Duplex  : Link duplex:
*                * 0: Duplex unknown or Auto-Neg. incomplete.
*                * 1: Half-Duplex.
*                * 2: Full-Duplex
*    Speed   : Link speed:
*                * == 0: Unknown, typically link-DOWN.
*                * >  0: Speeds of up to one gigabit are returned in Hertz.
*/
static void _ACD_cbOnLinkChange(unsigned IFaceId, U32 Duplex, U32 Speed) {
  IP_USE_PARA(Duplex);

  //
  // This is NOT our primary ACD monitored interface ?
  //
  if (IFaceId != _ACD_IFaceId) {
    //
    // This is a link-UP event ?
    //
    if (Speed != 0u) {
      //
      // Our ACD monitored interface is in background probing mode ?
      //
      if (_ACD_State == IP_ACD_STATE_ACTIVE_SEND_BG_PROBES) {
        //
        // Prepare to execute SemiActiveProbe
        //   - 200ms
        //   - Probe
        //   - 200ms
        //   - Probe
        //   - Back to background probing.
        //
        _ACD_SemiActiveProbesLeft = 2u;
        IP_ACD_UpdateBackgroundPeriod(_ACD_IFaceId, 200u);
      }
    }
  }
}

/*********************************************************************
*
*       MainTask
*/
void MainTask(void) {
  ...
  IP_Init();
  IP_AddLinkChangeHook(&_ACD_LinkChangeHook, _ACD_cbOnLinkChange);
```



```
  ...
  //
  // Wait for Link-UP.
  //
  ...
  IP_ACD_ActivateEx(_ACD_IFaceId, _ACD_cbOnInfo, &_ACD_Config, 0u);
  ...
}
```


14.5 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.

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
emNet ACD module	approximately 0.4 kByte

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
emNet ACD module	approximately 0.4 kByte

14.5.3 RAM usage

Addon	RAM
emNet ACD module	approximately 50 Bytes

Chapter 15

UPnP (Add-on)

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

15.1 emNet UPnP

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

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

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

The following table shows the contents of the emNet root directory:

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

15.2 Feature list

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

15.3 Requirements

TCP/IP stack

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

15.4 Backgrounds

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

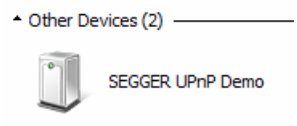
UPnP can be used to advertise services provided by a device across the network such as where to find the web interface for the device advertising. Newer operating systems support UPnP from scratch and will show UPnP devices available across a network and may provide easy access to a device by simply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

15.4.1 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.

```
/* Excerpt from OS_IP_Webserver_UPnP.c */
//
// UPnP
//
#define UPNP_FRIENDLY_NAME     "SEGGER UPnP Demo"
#define UPNP_MANUFACTURER      "SEGGER Microcontroller GmbH"
#define UPNP_MANUFACTURER_URL  "http://www.segger.com"
#define UPNP_MODEL_DESC        "SEGGER emWeb server with UPnP"
#define UPNP_MODEL_NAME        "SEGGER UPnP Demo"
#define UPNP_MODEL_URL         "http://www.segger.com/emweb"
```

The sample uses VFile hooks as described in `IP_WEBS_AddVFileHook()` 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 Webserver_DynContent.c */
//
// UPnP
//
#define UPNP_FRIENDLY_NAME     "SEGGER UPnP Demo"
#define UPNP_MANUFACTURER      "SEGGER Microcontroller GmbH"
#define UPNP_MANUFACTURER_URL  "http://www.segger.com"
#define UPNP_MODEL_DESC        "SEGGER emWeb server with UPnP"
#define UPNP_MODEL_NAME        "SEGGER UPnP Demo"
#define UPNP_MODEL_URL         "http://www.segger.com/emweb"

/* 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
*   IFaceId        - Zero-based interface index.
*   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(unsigned IFaceId, 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(IFaceId), 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
*   IFaceId        - Zero-based interface index.
*   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(unsigned IFaceId, 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(IFaceId, aHWAddr, sizeof(aHWAddr));
    SEGGER_snprintf(pBuffer,
                    NumBytes,
                    "%02X%02X%02X%02X%02X%02X",
                    aHWAddr[0],
                    aHWAddr[1],
                    aHWAddr[2],
                    aHWAddr[3],
                    aHWAddr[4],
                    aHWAddr[5]);
  }
  return pBuffer;
}

/*********************************************************************
*
*       _UPnP_GetSN
*
* Function description
*   This function copies the information needed for the SerialNumber parameter
*   into the given buffer and returns a pointer to the start of the buffer
*   for easy readable code.
*
* Parameters
*   IFaceId        - Zero-based interface index.
*   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(unsigned IFaceId, 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(IFaceId, aHWAddr, sizeof(aHWAddr));
    SEGGER_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
*   IFaceId        - Zero-based interface index.
*   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(unsigned IFaceId, 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(IFaceId, aHWAddr, sizeof(aHWAddr));
    SEGGER_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
*   IFaceId        - Zero-based interface index.
*   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(unsigned IFaceId,
                                            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(IFaceId), 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
*   IFaceId        - Zero-based interface index.
*   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(unsigned IFaceId,
                                        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);

  pf(pContextIn,   "<URLBase>", 0);
  pf(pContextIn,     _UPnP_GetURLBase(IFaceId, ac, sizeof(ac)), 0);
  pf(pContextIn,   "</URLBase>\r\n", 0);

  pf(pContextIn,   "<device>\r\n"
                     "<deviceType>urn:schemas-upnp-org:device:Basic:1</deviceType>
                      \r\n", 0);
  pf(pContextIn,     "<friendlyName>", 0);
  pf(pContextIn,       _UPnP_GetFriendlyName(IFaceId, ac, sizeof(ac)), 0);
  pf(pContextIn,     "</friendlyName>\r\n", 0);
  pf(pContextIn,     "<manufacturer>" UPNP_MANUFACTURER "</manufacturer>\r\n", 0);
  pf(pContextIn,     "<manufacturerURL>" UPNP_MANUFACTURER_URL "</manufacturerURL>
                      \r\n", 0);
  pf(pContextIn,     "<modelDescription>" UPNP_MODEL_DESC "</modelDescription>
                      \r\n", 0);
```



```
  pf(pContextIn,     "<modelName>" UPNP_MODEL_NAME "</modelName>\r\n", 0);

  pf(pContextIn,     "<modelNumber>", 0);
  pf(pContextIn,       _UPnP_GetModelNumber(IFaceId, ac, sizeof(ac)), 0);
  pf(pContextIn,     "</modelNumber>\r\n", 0);

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

  pf(pContextIn,     "<serialNumber>", 0);
  pf(pContextIn,       _UPnP_GetSN(IFaceId, ac, sizeof(ac)), 0);
  pf(pContextIn,     "</serialNumber>\r\n", 0);

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

  pf(pContextIn,     "<presentationURL>", 0);
  pf(pContextIn,       _UPnP_GetPresentationURL(IFaceId, ac, sizeof(ac)), 0);
  pf(pContextIn,     "</presentationURL>\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 Webserver_DynContent.c */
/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

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

  //
```



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

/*********************************************************************
*
*       _UPnP_SendVFile
*
* Function description
*   Send the content for the requested file using the callback provided.
*
* Parameters
*   pContextIn     - Send context of the connection processed for
*                    forwarding it to the callback used for output.
*   Index          - Index of the entry of a filename<=>content list
*                    if used. Alternative all comparisons can be done
*                    using the filename. In this case the index is
*                    meaningless. If using a filename<=>content list
*                    this is faster than searching again.
*   sFileName      - Name of the file that is requested. In case of
*                    working with the Index this is meaningless.
*   pf             - Function pointer to the callback that has to be
*                    for sending the content of the VFile.
*     pContextOut    - Out context of the connection processed.
*     pData          - Pointer to the data that will be sent
*     NumBytes       - Number of bytes to send from pData. If NumBytes
*                      is passed as 0 the send function will run a strlen()
*                      on pData expecting a string.
*/
static void _UPnP_SendVFile(void * pContextIn,
                            unsigned Index,
                            const char * sFileName,
                            void (*pf) (void * pContextOut,
                                        const char * pData,
                                        unsigned NumBytes)) {
  struct sockaddr_in LocalAddr;
         U32         IPAddr;
         long        hSock;
         int         IFaceId;
         int         Len;

  (void)sFileName;

  //
  // Generated VFiles
  //
  if (strcmp(sFileName, "/upnp.xml") == 0) {
    //
    // Retrieve socket that is used by connection.
    //
    hSock   = (long)IP_WEBS_GetConnectInfo((WEBS_OUTPUT*)pContextIn);
    Len     = sizeof(LocalAddr);
    getsockname(hSock, (struct sockaddr*)&LocalAddr, &Len);
    IPAddr  = ntohl(LocalAddr.sin_addr.s_addr);
    IFaceId = IP_FindIFaceByIP(&IPAddr, sizeof(IPAddr));
    if (IFaceId >= 0) {  // Only send back if we have found the interface.
      _UPnP_GenerateSend_upnp_xml(IFaceId, pContextIn, pf);
    }
    return;
  }
  //
  // Static VFiles
  //
```



```
  pf(pContextIn, _VFileList[Index].pData, _VFileList[Index].NumBytes);
}
```

All that is needed to be added to your application in order to provide the necessary XML files through emNet 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);
```


15.5 API functions

Function	Description
IP_UPNP_Activate()	Activates UPnP by joining an IGMP group and advertising that we are now available with services.

15.5.1 IP_UPNP_Activate()

Description

Activates UPnP by joining an IGMP group and advertising that we are now available with services.

Prototype

```
int IP_UPNP_Activate(      unsigned   IFaceId,
                     const char     * sUDN);
```

Parameters

Parameter	Description
IFaceId	Zero-base interface index.
sUDN	String containing a unique descriptor name. (Optional, can be NULL.)

Return value

= 0 O.K.
≠ 0 Error, UPnP not started.

Additional information

If sUDN is NULL, the unique descriptor name will be generated from the HW address of the interface.

15.6 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.

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

15.6.1 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
emNet UPnP	approximately 2.2 kByte

15.6.2 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
emNet UPnP	approximately 2.0 kByte

15.6.3 RAM usage

Addon	RAM
emNet UPnP	approximately 170 Bytes

Chapter 16

VLAN

The emNet 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.

16.1 emNet VLAN

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

16.2 Feature list

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

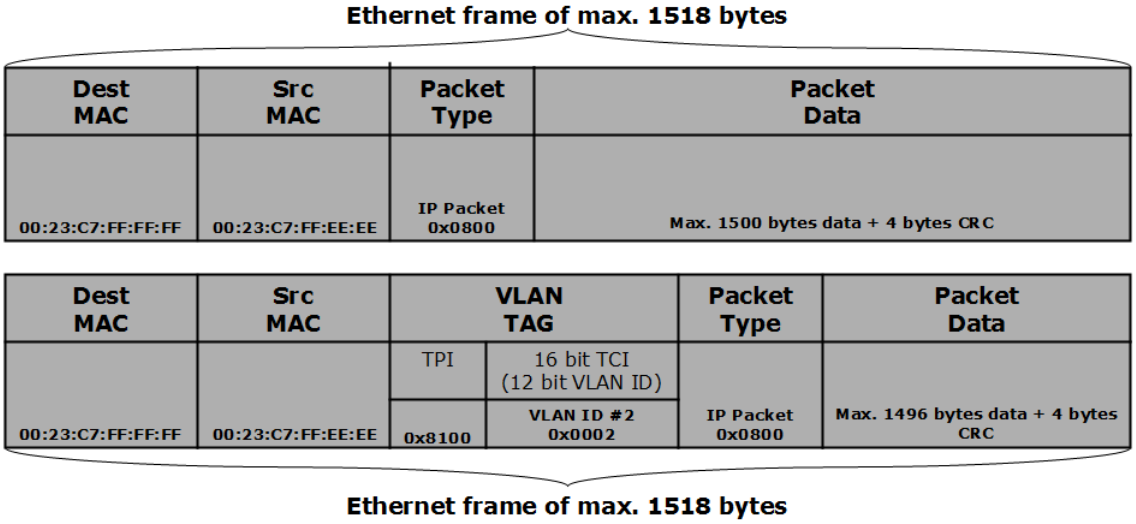
16.3 Backgrounds

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.



16.4 API functions

Function	Description
IP_VLAN_AddInterface()	Adds a VLAN interface to the stack.

16.4.1 IP_VLAN_AddInterface()

Description

Adds a VLAN interface to the stack.

Prototype

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

Parameters

Parameter	Description
HWIFace	Zero-based index of an available network interface to be used as physical interface for the VLAN pseudo interface.
VLANId	12 bits VLAN ID that the new interface will recognize. The priority bits can be set here as well. They will be included when sending packets on this interface. The priority bits for received packets are ignored.

Return value

≥ 0 Zero-based interface index of the newly created interface.
< 0 Error.

Additional information

Optional configuration of the maximum number of interfaces that can be added to the system using `IP_ConfigMaxIFaces()` needs to be done before adding any interface and must not be changed later.

16.5 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.

16.5.1 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
emNet VLAN	approximately 1.2 kByte

16.5.2 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
emNet VLAN	approximately 1.0 kByte

16.5.3 RAM usage

Addon	RAM
emNet VLAN	approximately 16 Bytes

Chapter 17

Tail Tagging (Add-on)

The emNet support for the Micrel Tail Tagging feature that is available in many Micrel Switch PHYs is an optional extension to emNet. 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.

17.1 emNet Tail Tagging support

The emNet 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 emNet even with different hardware addresses.

The following table shows the contents of the emNet root directory:

Directory	Content
BSP	Contains sample configurations for hardware that already uses emNet 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.

17.2 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 emNet stack.

17.3 Use cases

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 building hardware and software with special requirements. Some use cases are:

- Building a multhoming 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.

17.4 Requirements

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

17.4.1 Software requirements

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

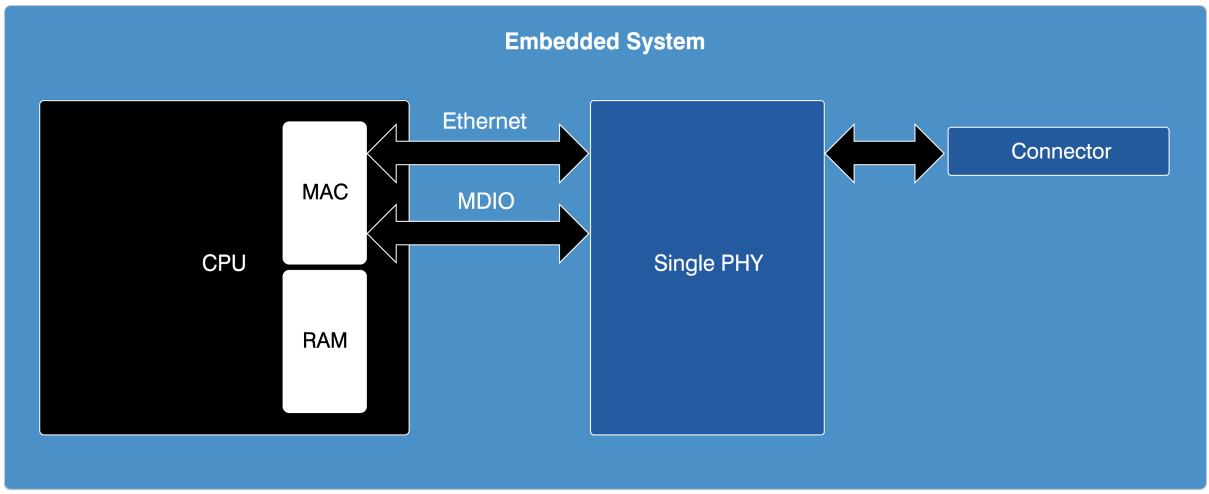
17.4.2 Hardware requirements

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.

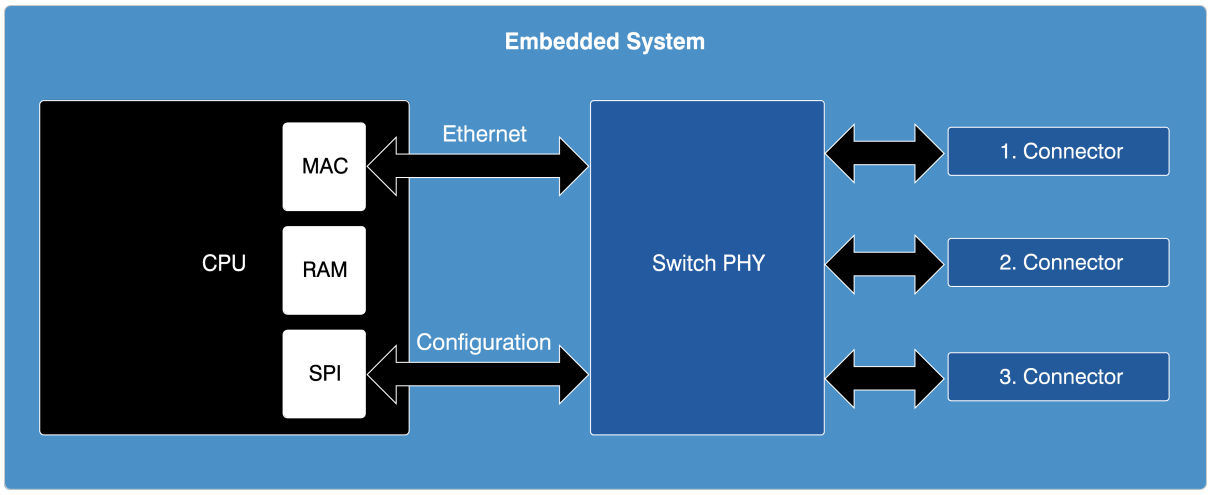
The Ethernet data is transferred between MAC and PHY while the MDIO interface (typically also accessed via registers of the MAC) is used to access the PHY to configure it and periodically check the link status.



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.



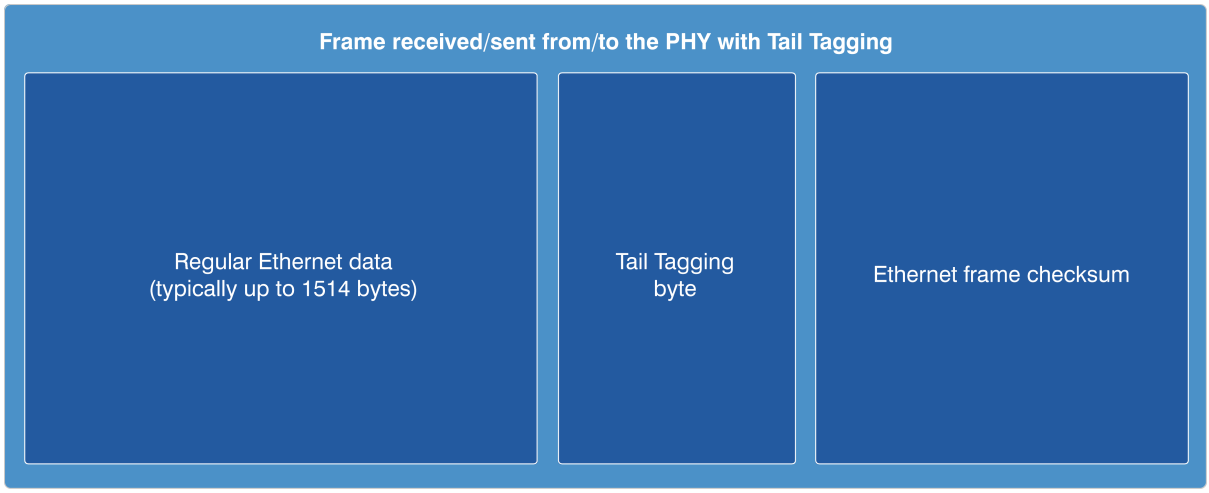
Using a Switch PHY with Tail Tagging not only allows you to connect multiple hosts but also allows you to fully control each external port/connector like it would be an additional expensive and external Ethernet controller.

17.5 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.

emNet 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 if 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.

17.6 Optimal MTU and buffer sizes

A Tail Tagging interface in emNet 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()`.

17.7 API functions

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

17.7.1 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 HWIFaceId,
                                        U8       InTag,
                                        U8       OutTag);
```

Parameters

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.

Return value

≥ 0 Zero-based interface index of the newly created interface.
< 0 Error.

Additional information

Optional configuration of the maximum number of interfaces that can be added to the system using `IP_ConfigMaxIFaces()` needs to be done before adding any interface and must not be changed later.

Example

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : IP_Config_K66_SEGGER_embOS_IP_SwitchBoard_ETH.c
Purpose     : Configuration file for TCP/IP with Freescale Kinetis K66
*/

#include "IP.h"
#include "IP_NI_KINETIS.h"
#include "BSP_IP.h"
#include "IP_PHY_MICREL_SWITCH.h"

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

#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\xDD\xFF\x22"  // MAC addr. used for target.
#define NUM_PORTS    3
  // Number of switch ports used for Tail-Tagging. 0: Plain switch mode.
```



```
#if (NUM_PORTS == 0)  // Keep memory for one port.
#define ALLOC_SIZE   (1         * 0x6000)
  // Size of memory dedicated to the stack in bytes.
#else
#define ALLOC_SIZE   (NUM_PORTS * 0x6000)
  // Size of memory dedicated to the stack in bytes. Very rough calculation.
#endif

/*********************************************************************
*
*       Defines, fixed
*
**********************************************************************
*/

#define SIM_SCGC5                     (*(volatile U32 *)(0x40048038))
   // System Clock Gating Control Register 5
#define SIM_SCGC6                     (*(volatile U32 *)(0x4004803C))
   // System Clock Gating Control Register 6
#define SIM_SCGC6_SPI0_MASK           (1uL << 12)
#define SIM_SCGC6_SPI1_MASK           (1uL << 13)
#define SIM_SCGC5_PORTB_MASK          (1uL << 10)
#define SIM_SCGC5_PORTC_MASK          (1uL << 11)

#define PORTB_BASE_ADDR               (0x4004A000)
#define PORTB_PCR10                   (*(volatile U32 *)(PORTB_BASE_ADDR + 0x0028))
   //  Pin Control Register 10
#define PORTB_PCR11                   (*(volatile U32 *)(PORTB_BASE_ADDR + 0x002C))
   //  Pin Control Register 11
#define PORTB_PCR16                   (*(volatile U32 *)(PORTB_BASE_ADDR + 0x0040))
   //  Pin Control Register 16
#define PORTB_PCR17                   (*(volatile U32 *)(PORTB_BASE_ADDR + 0x0044))
   //  Pin Control Register 17

#define PORTC_BASE_ADDR               (0x4004B000)
#define PORTC_PCR4                    (*(volatile U32 *)(PORTC_BASE_ADDR + 0x0010))
   //  Pin Control Register 4
#define PORTC_PCR5                    (*(volatile U32 *)(PORTC_BASE_ADDR + 0x0014))
   //  Pin Control Register 5
#define PORTC_PCR6                    (*(volatile U32 *)(PORTC_BASE_ADDR + 0x0018))
   //  Pin Control Register 6
#define PORTC_PCR7                    (*(volatile U32 *)(PORTC_BASE_ADDR + 0x001C))
   //  Pin Control Register 7

#define SPI0_BASE_ADDR                (0x4002C000)
#define SPI1_BASE_ADDR                (0x4002D000)
#define SPI_MCR                       (*(volatile U32 *)(SPI1_BASE_ADDR + 0x00))
#define SPI_CTAR0                     (*(volatile U32 *)(SPI1_BASE_ADDR + 0x0C))
#define SPI_SR                        (*(volatile U32 *)(SPI1_BASE_ADDR + 0x2C))
#define SPI_PUSHR                     (*(volatile U32 *)(SPI1_BASE_ADDR + 0x34))
#define SPI_POPR                      (*(volatile U32 *)(SPI1_BASE_ADDR + 0x38))

#define SPI_PUSHR_CONT_BIT            (1uL << 31)
   // Continued CS
#define SPI_PUSHR_EOQ_BIT             (1uL << 27)
   // End of queue
#define SPI_PUSHR_PCS_BIT             (1uL << 16)
   // Activate CS#0
#define SPI_MCR_HALT_BIT              (1uL <<  0)
   // Halt bit
#define SPI_SR_EOQF_BIT               (1uL << 28)
   // End of queue full in SR
#define SPI_SR_TCF_BIT                (1uL << 31)
   // Transfer complete
#define SPI_SR_RFDF_BIT               (1uL << 17)
   // RX Fifo drain flag

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

#ifdef __ICCARM__
```



```
  static __no_init U32 _aPool[ALLOC_SIZE / 4];
#else
//  #if (defined(__GNUC__) || defined(__SEGGER_CC__))
//    static U32 _aPool[ALLOC_SIZE / 4] __attribute__ ((section
 ("IP_RAM"))); // This is the memory area used by the stack.
//  #else
    static U32 _aPool[ALLOC_SIZE / 4];
//  #endif
#endif

/*********************************************************************
*
*       Local functions
*
**********************************************************************
*/

/*********************************************************************
*
*       _InitPhyIF()
*
*  Function description
*    Initializes the interface to the switch.
*/
static void _InitPhyIF(void) {
  U32 v;

  v          = SIM_SCGC5;
  v         |= SIM_SCGC5_PORTB_MASK; // Enable clock for Port B
  SIM_SCGC5  = v;
  v          = SIM_SCGC6;
  v         |= SIM_SCGC6_SPI1_MASK;  // Enable clock for SPI1
  SIM_SCGC6  = v;
  //
  // Set PTB10 (SPI1_PCS0) alternate function 2.
  //
  PORTB_PCR10 = 0x00000200;
  //
  // Set PTB11 (SPI1_SCK) alternate function 2.
  //
  PORTB_PCR11 = 0x00000200;
  //
  // Set PTB16 (SPI1_SOUT) alternate function 2.
  //
  PORTB_PCR16 = 0x00000200;
  //
  // Set PTB17 (SPI1_SIN) alternate function 2.
  //
  PORTB_PCR17 = 0x00000200;
  //
  // Setup SPI parameters.
  //
  SPI_MCR    = 0
             | (1uL << 31)  // Master mode
             | (1uL << 27)  // Freeze in debug mode
             | (1uL << 16)  // 1 = The inactive state of Peripheral chip select is high
             | (0uL << 13)  // 0 = TX FIFO is enabled
             | (0uL << 12)  // 0 = RX FIFO is enabled
             | (1uL <<  0)  // 0 = Start transfer
             ;
  SPI_CTAR0  = 0
             | (0uL << 31)  // Double baud rate
             | (7uL << 27)  // Frame size (7 + 1)
             | (0uL << 26)  // CPOL
             | (0uL << 25)  // CPHA
             | (0uL << 24)  // 0 = MSB first
             | (1uL << 22)  // PCSSCK
             | (1uL << 20)  // PASC
             | (3uL << 12)  // CSSCK
             | (3uL <<  0)  // Baud rate scaler
             ;
  //
  // Grant switch some time to completely power up.
  //
  IP_OS_Delay(100);
}
```



```
/*********************************************************************
*
*       _ReadWriteSPIByte()
*
*  Function description
*    Writes one byte via SPI and receives one byte in exchange.
*
*  Parameters
*    Data: Byte to write on line + settings.
*
*  Return value
*    Byte read from line.
*/
static U8 _ReadWriteSPI(U32 Data) {
  U8 v;

  SPI_PUSHR = SPI_PUSHR_PCS_BIT | Data;      // Push data + activation of CS0
  while ((SPI_SR & SPI_SR_TCF_BIT) == 0x0);  // Wait for transfer complete indication
  SPI_SR |= SPI_SR_TCF_BIT;                  // Reset transfer complete indication
  v = SPI_POPR;                              // Pop the read queue
  //
  return v;
}

/*********************************************************************
*
*       _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);

  SPI_MCR &= ~SPI_MCR_HALT_BIT;              // Activate SPI
  while ((SPI_SR & (1uL << 30)) == 0x0);     // Wait for ready indication
  //
  _ReadWriteSPI(      0
                    | SPI_PUSHR_CONT_BIT     // Continuous CS signal
                    | (0x03 << 5) );         // Read command
  _ReadWriteSPI(      0
                    | SPI_PUSHR_CONT_BIT     // Continuous CS signal
                    | (Reg << 1) );          // Register to read
  v = _ReadWriteSPI(  0
                    | SPI_PUSHR_EOQ_BIT      // End of Queue to transmit
                    | 0xff );                // Any value
  //
  SPI_MCR |= SPI_MCR_HALT_BIT;               // Halt SPI
  //
  return (v & 0xFF);
}

/*********************************************************************
*
*       _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);
```



```
  SPI_MCR &= ~SPI_MCR_HALT_BIT;              // Activate SPI
  while ((SPI_SR & (1uL << 30)) == 0x0);     // Wait for ready indication
  //
  _ReadWriteSPI(  0
                | SPI_PUSHR_CONT_BIT         // Continuous CS signal
                | (0x02 << 5) );             // Write command
  _ReadWriteSPI(  0
                | SPI_PUSHR_CONT_BIT         // Continuous CS signal
                | (Reg  << 1) );             // Register to read
  _ReadWriteSPI(  0
                | SPI_PUSHR_EOQ_BIT          // End of Queue to transmit
                | (v & 0xFF) );              // Value to write
  //
  SPI_MCR |= SPI_MCR_HALT_BIT;               // Halt SPI
}

/*********************************************************************
*
*       _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 (NUM_PORTS == 0)
  IP_USE_PARA(IFaceId);
#else
  if (IFaceId == 0) {  // Host interface ?
    //
    // Activate Tail Tagging. Needs to be done for the interface of the
    // host port. Enabling it multiple times does not hurt.
    //
    IP_PHY_MICREL_SWITCH_ConfigTailTagging(IFaceId, 1);  // 0: Off, 1: On.
  } else {
    //
    // 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.
    // This should be the first configuration to set as other functions might depend
    // on the port number set here internally.
    //
    IP_PHY_MICREL_SWITCH_AssignPortNumber(IFaceId, IFaceId - 1);
    //
    // Tx switch functionality and switch address learning.
    // For our Tail Tagging implementation for port multiplication we
    // want to disable the switch functionality for Tx as this would
    // send back incoming packets form one port to another creating
    // an infinite loop if both ports are in the same network.
    //
    IP_PHY_MICREL_SWITCH_ConfigRxEnable(IFaceId, 1);
    IP_PHY_MICREL_SWITCH_ConfigTxEnable(IFaceId, 0);
    IP_PHY_MICREL_SWITCH_ConfigLearnDisable(IFaceId, 1);
  }
#endif
}

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

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

/*********************************************************************
*
*       Global functions
*
```



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

/*********************************************************************
*
*       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                         // 4KBytes RAM.
*     mtu = 576;                                        // 576 is minimum acc. to RFC, 1500
 is max. for Ethernet.
*     IP_SetMTU(0, mtu);                                // Maximum Transmission Unit is 1500
 for Ethernet by default.
*     IP_AddBuffers(4, 256);                            // Small buffers.
*     IP_AddBuffers(2, mtu + 16);                       // Big buffers. Size should be mtu +
 16 byte for Ethernet header (2 bytes type, 2*6 bytes MAC, 2 bytes padding).
*     IP_ConfTCPSpace(2 * (mtu - 40), 1 * (mtu - 40));  // Define the TCP Tx and Rx window
 size. At least Tx space for 2*(mtu-40) for two full TCP packets is needed.
*
*   Microcontroller system, size optimized
*     #define ALLOC_SIZE 0x3000                         // 12KBytes RAM.
*     mtu = 576;                                        // 576 is minimum acc. to RFC, 1500
 is max. for Ethernet.
*     IP_SetMTU(0, mtu);                                // Maximum Transmission Unit is 1500
 for Ethernet by default.
*     IP_AddBuffers(8, 256);                            // Small buffers.
*     IP_AddBuffers(4, mtu + 16);                       // Big buffers. Size should be mtu +
 16 byte for Ethernet header (2 bytes type, 2*6 bytes MAC, 2 bytes padding).
*     IP_ConfTCPSpace(2 * (mtu - 40), 2 * (mtu - 40));  // Define the TCP Tx and Rx window
 size. At least Tx space for 2*(mtu-40) for two full TCP packets is needed.
*
*   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 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. At least Tx space for 2*(mtu-40) for two full TCP packets is needed.
*
*   System with lots of RAM
*     #define ALLOC_SIZE 0x20000                        // 128 KBytes RAM.
*     mtu = 1500;                                       // 576 is minimum acc. to RFC, 1500
 is max. for Ethernet.
*     IP_SetMTU(0, mtu);                                // Maximum Transmission Unit is 1500
 for Ethernet by default.
*     IP_AddBuffers(50, 256);                           // Small buffers.
*     IP_AddBuffers(50, mtu + 16);                      // Big buffers. Size should be mtu +
 16 byte for Ethernet header (2 bytes type, 2*6 bytes MAC, 2 bytes padding).
*     IP_ConfTCPSpace(8 * (mtu - 40), 8 * (mtu - 40));  // Define the TCP Tx and Rx window
 size. At least Tx space for 2*(mtu-40) for two full TCP packets is needed.
*/
void IP_X_Config(void) {
  int  mtu;
  int  IFaceId;
  int  HWIFaceId;
#if (NUM_PORTS != 0)
  int  i;
#endif
  U8   abHWAddr[6];
  char acTargetName[sizeof(TARGET_NAME)];

  _InitPhyIF();
  // Initialize the interface for the switch configuration.
  IP_AssignMemory(_aPool, sizeof(_aPool));
  // Assigning memory should be the first thing.
  IP_ConfigMaxIFaces(NUM_PORTS + 1);
  // Configure max. number of ports to be available.
  HWIFaceId = IP_AddEtherInterface(DRIVER);        // Add driver for your hardware.
```



```
#if (NUM_PORTS == 0)
  IFaceId   = HWIFaceId;
#endif
  IP_BSP_SetAPI(HWIFaceId, &BSP_IP_Api);
  // Set BSP callbacks for hardware access. Only required for HW interface.
  IP_NI_ConfigPHYMode(HWIFaceId, 1);               // Configure PHY Mode: 0: MII,
 1: RMII; For required hardware changes for RMII, please refer to your board manual
  //
  // Add PHY driver for the host port of the switch.
  // The PHY driver for the host port manages global
  // configurations like filter settings and other
  // things that are not setup for each port separately.
  //
  IP_PHY_AddDriver(HWIFaceId, &IP_PHY_Driver_Micrel_Switch_KSZ8895_HostPort, &PhyAccess, &_ConfigPHY);
  //
  // 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
                  | IP_MTYPE_APPLICATION     // Output application messages.
                  | IP_MTYPE_INIT            // Output all messages from init.
                  | IP_MTYPE_LINK_CHANGE     // Output a message if link status changes.
                  | IP_MTYPE_PPP             // Output all PPP/PPPoE related messages.
                  | IP_MTYPE_DHCP            // Output general DHCP status messages.
#if IP_SUPPORT_IPV6
                  | IP_MTYPE_IPV6            // Output IPv6 address related messages
#endif
//                  | IP_MTYPE_DHCP_EXT        // Output additional DHCP messages.
//                  | IP_MTYPE_CORE            // Output log messages from core module.
//                  | IP_MTYPE_ALLOC           // Output log messages for memory allocation.
//                  | IP_MTYPE_DRIVER          // Output log messages from driver.
//                  | IP_MTYPE_ARP             // Output log messages from ARP layer.
//                  | IP_MTYPE_IP              // Output log messages from IP layer.
//                  | IP_MTYPE_TCP_CLOSE
  // Output a log messages if a TCP connection has been closed.
//                  | IP_MTYPE_TCP_OPEN
  // Output a log messages if a TCP connection has been opened.
//                  | IP_MTYPE_TCP_IN          // Output TCP input logs.
//                  | IP_MTYPE_TCP_OUT         // Output TCP output logs.
//                  | IP_MTYPE_TCP_RTT         // Output TCP round trip time (RTT) logs.
//                  | IP_MTYPE_TCP_RXWIN       // Output TCP RX window related log messages.
//                  | IP_MTYPE_TCP             // Output all TCP related log messages.
//                  | IP_MTYPE_UDP_IN          // Output UDP input logs.
//                  | IP_MTYPE_UDP_OUT         // Output UDP output logs.
//                  | IP_MTYPE_UDP             // Output all UDP related messages.
//                  | IP_MTYPE_ICMP            // Output ICMP related log messages.
//                  | IP_MTYPE_NET_IN          // Output network input related messages.
//                  | IP_MTYPE_NET_OUT         // Output network output related messages.
//                  | IP_MTYPE_DNS             // Output all DNS related messages.
//                  | IP_MTYPE_SOCKET_STATE    // Output socket status messages.
//                  | IP_MTYPE_SOCKET_READ     // Output socket read related messages.
//                  | IP_MTYPE_SOCKET_WRITE    // Output socket write related messages.
//                  | IP_MTYPE_SOCKET          // Output all socket related messages.
                );
  //
  // 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);
  IP_MEMCPY(&acTargetName[0], TARGET_NAME, sizeof(TARGET_NAME));
```



```
#if (NUM_PORTS != 0)
  for (i = 0; i < NUM_PORTS; i++) {
    IFaceId = IP_MICREL_TAIL_TAGGING_AddInterface(HWIFaceId, i, (1 << 6) | (1 << i));
   // Add Tail Tagging interface for switch port, enable switch engine override.
#endif
    IP_SetHWAddrEx(IFaceId, (const U8*)&abHWAddr[0], 6);
   // Set MAC addr. for switch port: Needs to be unique for production units.
#if (NUM_PORTS != 0)
    abHWAddr[5]++;
   // Increase last byte of HW addr. for next switch port.
    acTargetName[sizeof(TARGET_NAME) - 1]++;
   // Increase last character of the target name for next switch port.

  IP_PHY_AddDriver(IFaceId, &IP_PHY_Driver_Micrel_Switch_KSZ8895, &PhyAccess, &_ConfigPHY);
  // Add PHY driver for Micrel switch PHY to the interface.
#endif
    IP_DHCPC_Activate(IFaceId, TARGET_NAME, NULL, NULL);
   // Activate DHCP client for this interface.
    //
    // Add IPv6 support to the stack and enable it for the interface.
    //
#if IP_SUPPORT_IPV6
    IP_IPV6_Add(IFaceId);
#endif
#if (NUM_PORTS != 0)
  }
#endif
  //
  // Run-time configure buffers.
  // The default setup will do for most cases.
  //
#if (NUM_PORTS == 0)
  IP_AddBuffers((1         * 12), 256);            // Small buffers.
  IP_AddBuffers((1         * 6), mtu + 16);        // Big buffers. Size should be mtu +
 16 byte for ethernet header (2 bytes type, 2*6 bytes MAC, 2 bytes padding)
#else
  IP_AddBuffers((NUM_PORTS * 12), 256);            // Small buffers.
  IP_AddBuffers((NUM_PORTS * 6), mtu + 16);        // Big buffers. Size should be mtu +
 16 byte for ethernet header (2 bytes type, 2*6 bytes MAC, 2 bytes padding)
#endif
  IP_ConfTCPSpace(3 * (mtu - 40), 3 * (mtu - 40)); // Define the TCP Tx and Rx window size
  IP_SOCKET_SetDefaultOptions(0
//                              | SO_TIMESTAMP
  // Send TCP timestamp to optimize the round trip time measurement. Normally not used in LAN.
                              | SO_KEEPALIVE
  // Enable keepalives by default for TCP sockets.
                             );
}

/****** End Of File *************************************************/
```


17.8 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.

17.8.1 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
emNet Tail Tagging	approximately 0.4 kByte

17.8.2 RAM usage

All required RAM is taken from the RAM that has been assigned to emNet using `IP_AddMemory()`. Only a few bytes are required.

Chapter 18

WiFi support

emNet WiFi support is an easy way to add the IEEE 802.11 standard also known as WiFi or WLAN to your project. It allows not only an easy start using WiFi in a new product but also allows adding WiFi support to existing projects already using emNet interfaces like LAN, PPP, USBD RNDIS/ECM or any other in short time.

All functions that are required to add WiFi to your application and some background information about WiFi hardware that can be used with emNet is described in this chapter.

18.1 emNet WiFi support

The emNet WiFi support allows adding different WiFi hardware in the same way as any other interface that can be added. The configuration and accessibility is bundled in an easy to understand API regardless of the underlying API of the WiFi hardware. This not only allows an easy start for adding WiFi to your project but comes in handy if you plan to exchange WiFi hardware in the future.

18.2 Feature list

- Unified API regardless of the WiFi module used.
- Easily add WiFi support as another interface to existing emNet LAN solutions.
- No need for re-certification by using already WiFi certified modules.
- Access Point support (depends on the module).
- Protocol support not limited to protocols that are TCP/UDP based.
- Support for various host interfaces like UART/SPI/SDIO/RMII (depends on the module).
- Low memory footprint (emNet WiFi software components).
- Seamless integration with the emNet stack.

18.3 Requirements

Software requirements

The emNet WiFi support requires the emNet TCP/IP stack, the emNet module (family) specific WiFi driver and typically a vendor SDK for the specific WiFi module (family).

The vendor SDK should feature a Hardware Abstraction Layer (HAL) that needs to be filled in with your interface specific hardware routines.

For WiFi modules that do not need an SDK, a HAL is provided by the emNet WiFi module specific driver, allowing to interface to various host interfaces supported by the WiFi module.

Hardware requirements

The emNet WiFi support can be used with virtually any module that is able to communicate with the host MCU and providing full Ethernet packet access. For communication with the host MCU typically an SPI interface is required. Other interfaces such as UART, SDIO, I2C or RMII might be supported by the module as well.

Different WiFi add-on boards for easy evaluation on the emPower eval board are available. The porting of the module specific hardware layer has already been done for these modules for the emPower eval board and are shipped with the drivers as example, can be found in the corresponding eval package.

18.4 Background information

This chapter does not cover the IEEE 802.11 standard as this would be too much information and required information about the standard itself can be easily found on the Internet. The background information referenced herein shall help to understand the level of implementation the emNet WiFi support offers.

18.4.1 Definition of a WiFi module

A WiFi module typically describes a small form factor board that basically consists of two components:

- An RF (Radio Frequency) module
- A companion MCU

WiFi modules are in fact external (Ethernet) controllers. The companion MCU is used to interface the RF module to actually establish WiFi communication. This is controlled by providing vendor specific commands to the companion MCU from the host MCU using one of the supported host interfaces of the WiFi module.

As each vendor is using its own command set, there is no common API that can be used across different modules, at least not between different vendors. Typically command sets are kept compatible within a product family of a vendor. This makes various modules compatible with the same SDK provided from the vendor.

Typically WiFi modules come with their own TCP/IP stack on board. This makes them easy to use for smaller projects like a weather station that periodically sends its data to a server. While these internal TCP/IP stacks might already come with some features like a small web server, they are also limited to the features of the built-in commands and `protocols.Benefits` of using WiFi modules

18.4.2 Benefits of using WiFi modules

While the WiFi circuitry could be directly integrated with your PCB there are various benefits from using an available WiFi module instead of your own implementation:

- No re-certification: Typically the WiFi modules available are already certified by the Wi-Fi Alliance. This saves a lot of time and costs otherwise spent for a certification process of your own designed WiFi circuitry.
- Using a design that has proven to be working.
- Easy to evaluate/no need for prototyping in regards to the WiFi circuitry.
- Can easily extend existing solutions using a free standard peripheral interface like SPI without redesigning the whole hardware.
- Offloading crypto operations necessary for encryption like WPA2 to the companion MCU.

Besides the hardware designing and evaluation aspects, the biggest benefit is without doubt the elimination of a re-certification process which might be time and cost intensive. Typically the modules are completely certified when using an integrated antenna or are certified when being used with a selection of one or more antennas specified by the module vendor.

18.4.3 Module internal vs. external TCP/IP stack

While most WiFi modules come with their own internal TCP/IP stack on the companion MCU, they are typically limited in usage to their built-in commands. This usually means only having access to a limited amount of TCP and UDP sockets that can be used to implement higher level protocols based on these two base protocols.

While only having a limited amount of TCP and UDP sockets might be enough for some small projects, this concept lacks control and extensibility. To allow more control and extensibility an external TCP/IP stack like emNet needs to be used. This allows having control over the complete Ethernet frame of the packet to implement protocols on a lower level such as ARP or VLAN.

It is often referred to as `pass-through-mode` or `bypass-mode` to give an external stack full control over the whole Ethernet frame. It disables the processing by the module internal TCP/IP stack and exchanges the complete Ethernet frame with the TCP/IP stack on the host MCU.

18.4.4 Supported WiFi modules

The intention of WiFi support for emNet is to allow extending an already established product with WiFi in a flexible and easy way. At the same time it shall be an easy to use solution for new projects. Features shall not depend on module features in the first place and shall be extensible at any time. emNet supports only modules that are able to exchange complete Ethernet frames with the host MCU using a so called `pass-through-mode` or `bypass-mode` to fulfill this goal.

Being able to access the whole Ethernet frame, emNet is not only able to use TCP and UDP based high level protocols but allows low level protocols via WiFi as well. Using this solution, existing and future add-ons can be used via WiFi the same way they would be with a cable based solution.

18.5 API functions

Function	Description
IP_WIFI_AddAssociateChangeHook()	Adds a function to the IP_HOOK_ON_WIFI_ASSOCIATE_CHANGE list.
IP_WIFI_AddClientNotificationHook()	Adds a function to the IP_HOOK_ON_WIFI_CLIENT_NOTIFICATION list.
IP_WIFI_AddInterface()	Adds a WiFi interface to the stack.
IP_DTASK_AddExecDoneHook()	Adds a callback that gets executed once the Driver Task handler is done.
IP_WIFI_AddSignalChangeHook()	Adds a function to the IP_HOOK_ON_WIFI_SIGNAL_CHANGE list.
IP_WIFI_ConfigAllowedChannels()	Configures the channels that are allowed to be used for network scan and associate requests.
IP_DTASK_ConfigAlwaysSignaled()	Keeps the interface in signaled state to be polled each time regardless if there really was a signal or not.
IP_DTASK_GetTimeout()	Retrieves the timeout [ms] after which the Driver Task should polls the driver interrupt routine even if it has not been signaled.
IP_DTASK_SetTimeout()	Sets the timeout [ms] after which the Driver Task polls the driver interrupt routine even if it has not been signaled.
IP_WIFI_Connect()	Connects to a selected SSID or starts as access point.
IP_WIFI_Disconnect()	Disconnects from any connected network or stops the access point mode.
IP_DTASK_Task()	Task that polls the handler routine of some drivers.
IP_DTASK_Init()	Initializes the DriverTask context.
IP_DTASK_Exec()	Executes the handler routine of the driver for a specific interface.
IP_DTASK_ExecAll()	Executes the handler routine of the driver for all interfaces.
IP_DTASK_WaitForEvent()	Waits for an event for the DriverTask to be signaled.
IP_WIFI_Scan()	Scans for available wireless networks.
IP_WIFI_Security2String()	Converts the numeric security value to a readable text.
IP_DTASK_Signal()	Signals the Driver Task to poll the handler routine of the driver.

18.5.1 IP_WIFI_AddAssociateChangeHook()

Description

Adds a function to the IP_HOOK_ON_WIFI_ASSOCIATE_CHANGE list.

Prototype

```
void IP_WIFI_AddAssociateChangeHook(IP_HOOK_ON_WIFI_ASSOCIATE_CHANGE * pHook,
                                    IP_WIFI_pfOnAssociateChange        pf);
```

Parameters

Parameter	Description
pHook	Pointer to hook structure to link.
pf	Pointer to function to call on change.

18.5.2 IP_WIFI_AddClientNotificationHook()

Description

Adds a function to the IP_HOOK_ON_WIFI_CLIENT_NOTIFICATION list. This list is notified when a client connects or disconnect when in access point mode.

Prototype

```
void IP_WIFI_AddClientNotificationHook(IP_HOOK_ON_WIFI_CLIENT_NOTIFICATION * pHook,
                                       IP_WIFI_pfOnClientNotification        pf);
```

Parameters

Parameter	Description
pHook	Pointer to hook structure to link.
pf	Pointer to function to call on notification.

18.5.3 IP_WIFI_AddInterface()

Description

Adds a WiFi interface to the stack.

Prototype

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

Parameters

Parameter	Description
pDriver	Pointer to IP_HW_DRIVER API table.

Return value

≥ 0 Zero-based interface index of the newly created interface.
< 0 Error.

Additional information

Optional configuration of the maximum number of interfaces that can be added to the system using IP_ConfigMaxIFaces() needs to be done before adding any interface and must not be changed later.

18.5.4 IP_DTASK_AddExecDoneHook()

Description

Adds a callback that gets executed once the Driver Task handler is done.

Prototype

```
void IP_DTASK_AddExecDoneHook(unsigned                      IFaceId,
                              IP_HOOK_ON_DTASK_EXEC_DONE * pHook,
                              IP_ON_DTASK_EXEC_DONE_FUNC * pf);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pHook	Pointer to hook structure to link.
pf	Pointer to function to call on change.

Additional information

A callback that signals the end of the ISR handler routine is required when using level sensitive interrupts that only signal a task like the WiFi/DTask ISR task to run. The following example demonstrates why this is necessary:

1. Interrupt line gets high.
2. Level sensitive interrupt is fired.
3. Signaling the WiFi ISR task to run.
4. Clearing the interrupt pending flag (could have been done before 3. as well).
5. The interrupt is still pending as typically the interrupt line on WiFi modules only gets low after all messages have been received. For this the WiFi/DTask ISR Task would need to run but we are stuck at 2. as the level sensitive interrupt constantly fires.

Solution:

After 2. simply disable the interrupt. Once all messages we are aware of have been processed the WiFi/DTask ISR task will run to wait until it is signaled again. Before actually waiting the callback gets executed telling us that now is the right moment to clear the pending interrupt flag and re-enabling the interrupt itself as most likely the interrupt line is now low and we are not instantly back in the interrupt. Of course it might happen that we are almost instantly back the interrupt as new messages are ready at the module to be received.

18.5.5 IP_WIFI_AddSignalChangeHook()

Description

Adds a function to the IP_HOOK_ON_WIFI_SIGNAL_CHANGE list.

Prototype

```
void IP_WIFI_AddSignalChangeHook(IP_HOOK_ON_WIFI_SIGNAL_CHANGE * pHook,
                                 IP_WIFI_pfOnSignalChange        pf);
```

Parameters

Parameter	Description
pHook	Pointer to hook structure to link.
pf	Pointer to function to call on change.

18.5.6 IP_WIFI_ConfigAllowedChannels()

Description

Configures the channels that are allowed to be used for network scan and associate requests.

Prototype

```
int IP_WIFI_ConfigAllowedChannels(      unsigned   IFaceId,
                                  const U8       * paChannel,
                                        U8         NumChannels);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
paChannel	Pointer to a list of allowed channels.
NumChannels	Number of channels in list.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

Allowed channels are a subset of the configured regulatory domain.

18.5.7 IP_DTASK_ConfigAlwaysSignaled()

Description

Keeps the interface in signaled state to be polled each time regardless if there really was a signal or not.

Prototype

```
void IP_DTASK_ConfigAlwaysSignaled(unsigned IFaceId,
                                   char     OnOff);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	• 0 : Off. • Other: On.

18.5.8 IP_DTASK_GetTimeout()

Description

Retrieves the timeout [ms] after which the Driver Task should polls the driver interrupt routine even if it has not been signaled.

Prototype

```
unsigned IP_DTASK_GetTimeout(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

Previously configured timeout [ms].

Additional information

At the moment the IFaceId parameter is ignored and the timeout value is used for all interfaces.

This routine can be used to set the timeout in a central place such as from IP_X_Config() and retrieve it wherever necessary.

18.5.9 IP_DTASK_SetTimeout()

Description

Sets the timeout [ms] after which the Driver Task polls the driver interrupt routine even if it has not been signaled.

Prototype

```
void IP_DTASK_SetTimeout(unsigned IFaceId,
                         unsigned Timeout);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Timeout	Timeout [ms].

Additional information

At the moment the IFaceId parameter is ignored and the timeout value is used for all interfaces.

18.5.10 IP_WIFI_Connect()

Description

Connects to a selected SSID or starts as access point.

Prototype

```
int IP_WIFI_Connect(      unsigned                 IFaceId,
                    const IP_WIFI_CONNECT_PARAMS * pParams,
                          U32                      Timeout);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pParams	Pointer to structure that contains connection parameters.
Timeout	Timeout before considering connect attempt failed [ms].

Return value

= 0 O.K.
≠ 0 Error.

18.5.11 IP_WIFI_Disconnect()

Description

Disconnects from any connected network or stops the access point mode.

Prototype

```
int IP_WIFI_Disconnect(unsigned IFaceId,
                       U32      Timeout);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Timeout	Timeout before considering disconnect attempt failed [ms].

Return value

= 0 O.K.
≠ 0 Error.

18.5.12 IP_DTASK_Task()

Description

Task that polls the handler routine of some drivers.

Prototype

```
void IP_DTASK_Task(void);
```

Additional information

This task is required to be implementing into your project for some drivers to work. This is typically the case for external Ethernet controllers. An example for typical task stack usage is defined by `TASK_STACK_SIZE_IP_DRIVER_TASK` .

For best performance this task should be given a task priority higher than any other IP stack related application task and even the `IP_Task()` or its API alternatives `IP_TASK_Init()`, `IP_TASK_Exec()` and `IP_TASK_WaitForEvent()` . It however must not have a higher or the same priority than the `IP_RxTask()` or its API alternatives `IP_RXTASK_Init()`, `IP_RXTASK_Exec()` and `IP_RXTASK_WaitForEvent()` .

For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

After startup, this routine settles into a loop, handling driver events. This loop sleeps until signaled by an event. Alternatively it can be configured to wake up and poll the drivers periodically using `IP_DTASK_ConfigTimeout()` and `IP_DTASK_ConfigAlwaysSignaled()` .

In case of de-initializing the stack with `IP_DeInit()`, it is possible to leave the loop gracefully by using `IP_ShutDown()` .

18.5.13 IP_DTASK_Init()

Description

Initializes the DriverTask context.

Prototype

```
void IP_DTASK_Init(void);
```

Additional information

> **Note**
>
> This routine is not intended to be used when using `IP_DTASK_Task()` instead. It needs to be called before `IP_DTASK_Exec()` or `IP_DTASK_WaitForEvent()` is used.

For best performance the `IP_DTASK_*` API should be called with a task priority higher than any other IP stack related application task and even the `IP_Task()` or its API alternatives `IP_TASK_Init()`, `IP_TASK_Exec()` and `IP_TASK_WaitForEvent()` .

> **Warning**
>
> The task priority from which this routine is executed must not be higher or the same priority than a task executing the `IP_RxTask()` or its API alternatives `IP_RXTASK_Init()`, `IP_RXTASK_Exec()` and `IP_RXTASK_WaitForEvent()` .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

Example

```
/*********************************************************************
*
*       _IP_DTASK_Task()
*
*  Function description
*    Application specific implementation of IP_DTASK_Task() .
*
*  Additional information
*    Allows to insert your own code like feeding a watchdog
*    in-between the separate steps that would be executed by the
*    original task API provided by the stack.
*/
static void _IP_DTASK_Task(void) {
  unsigned Timeout;

  //
  // Initialize.
  //
  IP_DTASK_Init();
  //
  // Get the timeout configured for example during IP_X_Config() .
  // If not configured, the default is returned which is 0 and
  // means to wait INFINITE .
  //
  Timeout = IP_DTASK_GetTimeout(0u);  // Get timeout for interface #0 .
  //
  // Task-loop.
  //
  for (;;) {
    //
    // Wait with timeout [ms] for the next event to be signaled.
```



```
    // Typically the signal is triggered by an interrupt from an
    // external controller that notifies us that the previous
    // operation is now finished and the next can be started.
    //
    IP_DTASK_WaitForEvent(Timeout);
    //
    // Process the event.
    //
    IP_DTASK_ExecAll();
  }
}
```


18.5.14 IP_DTASK_Exec()

Description

Executes the handler routine of the driver for a specific interface.

Prototype

```
void IP_DTASK_Exec(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface ID.

Additional information

This routine is an alternative to using the `IP_DTASK_Task()` . It allows finer control over the internal steps done in `IP_DTASK_Task()` . This can be utilized for example to feed a watchdog from the same task periodically.

> **Note**
>
> This routine is not intended to be used when using `IP_DTASK_Task()` instead.

For best performance the `IP_DTASK_*` API should be called with a task priority higher than any other IP stack related application task and even the `IP_Task()` or its API alternatives `IP_TASK_Init()`, `IP_TASK_Exec()` and `IP_TASK_WaitForEvent()` .

> **Warning**
>
> The task priority from which this routine is executed must not be higher or the same priority than a task executing the `IP_RxTask()` or its API alternatives `IP_RXTASK_Init()`, `IP_RXTASK_Exec()` and `IP_RXTASK_WaitForEvent()` .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

18.5.15 IP_DTASK_ExecAll()

Description

Executes the handler routine of the driver for all interfaces.

Prototype

```
void IP_DTASK_ExecAll(void);
```

Additional information

This routine is an alternative to using the `IP_DTASK_Task()` . It allows finer control over the internal steps done in `IP_DTASK_Task()` . This can be utilized for example to feed a watchdog from the same task periodically.

> **Note**
>
> This routine is not intended to be used when using `IP_DTASK_Task()` instead.

For best performance the `IP_DTASK_*` API should be called with a task priority higher than any other IP stack related application task and even the `IP_Task()` or its API alternatives `IP_TASK_Init()`, `IP_TASK_Exec()` and `IP_TASK_WaitForEvent()` .

> **Warning**
>
> The task priority from which this routine is executed must not be higher or the same priority than a task executing the `IP_RxTask()` or its API alternatives `IP_RX-TASK_Init()`, `IP_RXTASK_Exec()` and `IP_RXTASK_WaitForEvent()` .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

18.5.16 IP_DTASK_WaitForEvent()

Description

Waits for an event for the DriverTask to be signaled.

Prototype

```
unsigned IP_DTASK_WaitForEvent(unsigned Timeout);
```

Parameters

Parameter	Description
Timeout	Timeout [ms] to wait for an event. 0 for INFINITE .

Return value

= 0 An event was signaled.
≠ 0 Timeout.

Additional information

This routine is an alternative to using the IP_DTASK_Task() . It allows finer control over the internal steps done in IP_DTASK_Task() . This can be utilized for example to feed a watchdog from the same task periodically.

> **Note**
>
> This routine is not intended to be used when using IP_DTASK_Task() instead.

For best performance the IP_DTASK_* API should be called with a task priority higher than any other IP stack related application task and even the IP_Task() or its API alternatives IP_TASK_Init(), IP_TASK_Exec() and IP_TASK_WaitForEvent() .

> **Warning**
>
> The task priority from which this routine is executed must not be higher or the same priority than a task executing the IP_RxTask() or its API alternatives IP_RX-TASK_Init(), IP_RXTASK_Exec() and IP_RXTASK_WaitForEvent() .
>
> For more information regarding task priorities, please refer to *Tasks and interrupt usage* on page 45 .

18.5.17 IP_WIFI_Scan()

Description

Scans for available wireless networks.

Prototype

```
int IP_WIFI_Scan(      unsigned               IFaceId,
                       U32                    Timeout,
                       IP_WIFI_pfScanResult   pf,
                 const char                 * sSSID,
                       U8                     Channel);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Timeout	Timeout before aborting the scan [ms].
pf	Callback to be used for each single result.
sSSID	SSID to find. May be NULL to scan all available networks.
Channel	Selected channel to scan. 0 means all channels.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

If an SSID to find has been set the result callback will report the connection parameters only for the selected SSID. Without a given SSID a list of available networks and their parameters will be returned.

A network scan means that the module needs to set one of its antennas into monitoring mode, listening for beacon frames with SSIDs regularly sent by Access Points. If a module has only one antenna, a scan might not be possible while being connected, typically also returning an error for this API call.

18.5.18 IP_WIFI_Security2String()

Description

Converts the numeric security value to a readable text.

Prototype

```
char *IP_WIFI_Security2String(U8 Security);
```

Parameters

Parameter	Description
Security	Numeric security value.

Return value

Pointer to string of the security.

18.5.19 IP_DTASK_Signal()

Description

Signals the Driver Task to poll the handler routine of the driver.

Prototype

```
void IP_DTASK_Signal(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

18.6 Data structures

Structure / Callback	Description
IP_WIFI_CONNECT_PARAMS	Used to configure parameters for connecting to an Access Point or starting your own Access Point.

18.6.1 Structure IP_WIFI_CONNECT_PARAMS

Description

Used to configure parameters for connecting to an Access Point or starting your own Access Point.

Prototype

```
typedef struct {
  const char*             sSSID;
  const char*             sWPAPass;
  const IP_WIFI_WEP_KEY* paWEPKey;
  U8                      abBSSID[6];
  U8                      NumWEPKeys;
  U8                      WEPActiveKeyIndex;
  U8                      Mode;
  U8                      Security;
  U8                      Channel;
} IP_WIFI_CONNECT_PARAMS;
```

Member	Description
sSSID	SSID to connect to or to open when in Access point mode.
sWPAPass	WPA(2) passphrase to use.
paWEPKey	Array of pointers to binary WEP keys.
abBSSID	HW address of the access point to connect to.
NumWEPKeys	Number of WEP keys configured in paWEPKey.
WEPActiveKeyIndex	0..3: Index of WEP key to be used for sending, typically index 0 .
Mode	IP_WIFI_MODE_INFRASTRUCTURE or IP_WIFI_MODE_ACCESS_POINT (if supported by the driver and moodule).
Security	Security used or security to use if we are starting an Access Point. IP_WIFI_SECURITY_OPEN or IP_WIFI_SECURITY_WEP_OPEN or IP_WIFI_SECURITY_WEP_SHARED or IP_WIFI_SECURITY_WPA_TKIP or IP_WIFI_SECURITY_WPA_AES or IP_WIFI_SECURITY_WPA_WPA2_MIXED or IP_WIFI_SECURITY_WPA2_AES.
Channel	Channel to use for connect or starting an Access Point. When connecting 0 means any.

Chapter 19

Network interface drivers

emNet has been designed to cooperate with any kind of hardware. To use specific hardware with emNet, 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.

19.1 Network interface drivers general information

To use emNet, 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 581.

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.

19.1.1 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.

19.1.2 Checksum computation in hardware

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

19.1.3 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.

19.2 Available network interface drivers

Network interface drivers are optional components to emNet. Network interface drivers are already available for popular hardware and a list could be found on our website https://www.segger.com/emnet-drivers.

19.2.1 Configuring the driver

The interface with the corresponding driver should be added to the configuration function `IP_X_Config()` with a call to `IP_AddEtherInterface()`.

Depending on the hardware, it could be needed to configure the PHY mode for RMII with `IP_NI_ConfigPHYMode()`. As the default configuration is MII, the call is optional and requested only to use RMII.

Some drivers provides also configuration functions, for example to configure the number of buffer used, specify registers base address, …

19.2.2 BSP configuration

Drivers are calling BSP function in order to configure hardware access:

- Old driver are using functions like `BSP_ETH_Init()` and `BSP_ETH_InstallISR()` which are implemented in `BSP.c`.
- New and updated drivers are using the structure `BSP_IP_API` implemented in `BSP_IP.c`. See *Structure BSP_IP_API* on page 288 for more details. This structure needs to be set as BSP access API with the function `IP_BSP_SetAPI()`.

19.2.3 Driver configuration example

The following code is an excerpt of `IP_X_Config()`. Reference to *IP_X_Config* on page 616 for a complete example.

```
IFaceId = IP_AddEtherInterface(DRIVER);  // Add driver for your hardware.
IP_BSP_SetAPI(IFaceId, &BSP_IP_Api);
  // Set BSP callbacks for hardware access.
IP_SetHWAddrEx(IFaceId, (const U8*)HW_ADDR, 6);
  // MAC addr.: Needs to be unique.
IP_NI_ConfigPHYMode(IFaceId, 1);         // Configure PHY Mode: 0: MII, 1: RMII .
```


19.3 Writing your own driver

If you are going to use emNet 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 emNet.

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.

19.3.1 Network interface driver structure

emNet 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 {
  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);
  void  (*pfEnDisableRxInt)   (unsigned Unit, unsigned OnOff);
} IP_HW_DRIVER;
```

Elements of IP_HW_DRIVER

Element	Meaning
pfInit	Pointer to the initialization function.
pfSendPacket	Pointer to the send packet function.
pfGetPacketSize	Pointer to the get packet size function.
pfReadPacket	Pointer to the read packet function.
pfTimer	Optional: Pointer to the timer function. The routine is called from the stack periodically
pfControl	Pointer to the control function.
pfEnDisableRxInt	Optional: Pointer to enable/disable Rx interrupts.

Example

```
/* Sample implementation taken from the driver for FREESCALE Kinetis */

/*********************************************************************
*
*       Driver API Table
*
**********************************************************************
*/

const IP_HW_DRIVER IP_Driver_Kinetis = {
  _Init,
  _SendPacketIfTxIdle,
  _GetPacketSize,
  _ReadPacket,
  _Timer,
  _Control,
  _EnDisableRxInt
};
```


19.3.2 Device driver functions

This section provides descriptions of the network interface driver functions required by emNet. Note that the names used for these functions are not really relevant for emNet because the stack accesses them through a structure of function pointers.

Function	Description
`_Init()`	General initialization function of the driver.
`_SendPacketIfTxIdle()`	Send the next packet in the send queue if transmitter is idle.
`_GetPacketSize()`	Reads buffer descriptors to find out if a packet has been received.
`_ReadPacket()`	Reads the first packet in the buffer.
`_Timer()`	Timer function called by the networking task, `IP_Task()`, once per second.
`_Control()`	This function is used to implement additional driver specific control functions. It can be empty.
`_EnDisableRxInt()`	Utility function to enable or disable receive interrupts. It can be empty.

19.3.3 Driver template

The driver template `IP_NI_TEMPLATE.c` is supplied in the folder `Sample\IP\Driver\Template\`.

Example

```
/*********************************************************************
*               (c) SEGGER Microcontroller GmbH & Co. KG             *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

------------------------- END-OF-HEADER -----------------------------

File    : IP_NI_TEMPLATE.c
Purpose : Network interface driver template.
*/

#include "IP_Int.h"
#include "IP_NI_TEMPLATE.h"

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/

//None

/*********************************************************************
*
*       Defines, fixed
*
**********************************************************************
*/

#define MAC_ISR_STATUS           (*(volatile U32*)(0x00000000))

#define MAC_ISR_STATUS_RX_MASK   (1uL << 0)
#define MAC_ISR_STATUS_TX_MASK   (1uL << 1)

/*********************************************************************
*
*       Types, local
```



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

#if IP_DEBUG
typedef struct {
  U32 TxSendCnt;
  U32 TxIntCnt;
  U32 TxErrCnt;
  U32 RxIntCnt;
  U32 RxCnt;
  U32 RxErrCnt;
} IP_NI_STATS;

  #define INC(Cnt)  _Stats.Cnt++
#else
  #define INC(Cnt)
#endif

/*********************************************************************
*
*       Prototypes
*
**********************************************************************
*/

static unsigned _PHY_ReadReg (IP_PHY_CONTEXT* pContext, unsigned RegIndex);
static void     _PHY_WriteReg(IP_PHY_CONTEXT* pContext, unsigned RegIndex, unsigned v);

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static const IP_PHY_ACCESS _PHY_Access = {
  _PHY_ReadReg,   // pfRead
  _PHY_WriteReg   // pfWrite
};

static const IP_PHY_DRIVER* _pPHY_Driver = &IP_PHY_Generic;
  // Use generic PHY driver as default.
static       IP_PHY_CONTEXT _PHY_Context;
static       IP_NI_STATS    _Stats;
static       char           _TxIsBusy;
static       char           _IsInited;

/*********************************************************************
*
*       Local functions
*
**********************************************************************
*/

/*********************************************************************
*
*       _PHY_ReadReg()
*
*  Function description
*    Reads from a PHY register.
*
*  Parameters
*    pContext: Pointer to PHY context of IP_PHY_CONTEXT .
*    RegIndex: Register index to access.
*
*  Return value
*    Data read from PHY.
*/
static unsigned _PHY_ReadReg(IP_PHY_CONTEXT* pContext, unsigned RegIndex) {
  unsigned Addr;
  unsigned v;

  Addr = pContext->Addr;
  IP_USE_PARA(Addr);
  IP_USE_PARA(RegIndex);
```



```
  //
  // Read data from PHY.
  //
  v = 0;
  return v;
}

/*********************************************************************
*
*       _PHY_WriteReg()
*
*  Function description
*    Writes to a PHY register.
*
*  Parameters
*    pContext: Pointer to PHY context of IP_PHY_CONTEXT .
*    RegIndex: Register index to access.
*    v       : Data to write to register.
*/
static void _PHY_WriteReg(IP_PHY_CONTEXT* pContext, unsigned RegIndex, unsigned v) {
  unsigned Addr;

  Addr = pContext->Addr;
  IP_USE_PARA(Addr);
  IP_USE_PARA(RegIndex);
  //
  // Write data to PHY.
  //
  IP_USE_PARA(v);
}

/*********************************************************************
*
*       _SetFilter()
*
*  Function description
*    Sets the MAC filter(s).
*    The stack tells the driver which addresses should go through
*    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.
*
*  Parameters
*    pFilter: Filters to set.
*
*  Return value
*    == 0: O.K.
*
*  Additional information
*    In general, precise filtering is used as far as supported by
*    the hardware. If more addresses need to be filtered than
*    precise address filters are available, then a hash filter
*    should be used. Alternatively the MAC can be switched to
*    promiscuous mode for simple implementations.
*/
static int _SetFilter(IP_NI_CMD_SET_FILTER_DATA* pFilter) {
  const U8*      pAddrData;  // Pointer to 6 byte filter addresses.
        unsigned u;
        unsigned NumAddr;

  NumAddr   = pFilter->NumAddr;
  pAddrData = pFilter->pHWAddr;

  for (u = 0; u < NumAddr; u++) {
    //
    // Set filter.
    //
    pAddrData += 6;  // Proceed to next filter addr.
  }

  IP_USE_PARA(pAddrData);
  return 0;  // O.K.
}

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



```
*
*       _UpdateMACSettings()
*
*  Function description
*    Updates the speed & duplex settings of the MAC.
*    Needs to be called whenever speed and duplex settings change.
*
*  Parameters
*    Duplex : Duplex configuration to set:
*               * IP_DUPLEX_UNKNOWN
*               * IP_DUPLEX_HALF
*               * IP_DUPLEX_FULL
*    Speed  : Speed configuration to set:
*               * IP_SPEED_UNKNOWN
*               * IP_SPEED_10MHZ
*               * IP_SPEED_100MHZ
*               * IP_SPEED_1GHZ
*/
static void _UpdateMACSettings(U32 Duplex, U32 Speed) {
  IP_USE_PARA(Duplex);
  IP_USE_PARA(Speed);

  //
  // Update MAC configuration.
  //
}

/*********************************************************************
*
*       _UpdateLinkState()
*
*  Function description
*    Reads link state information from PHY and updates MAC if
*    necessary. Should be called regularly to make sure that the
*    MAC is notified if the link changes.
*/
static void _UpdateLinkState(void) {
  U32 Duplex;
  U32 Speed;

  _pPHY_Driver->pfGetLinkState(&_PHY_Context, &Duplex, &Speed);
  if (IP_NI_SetLinkState(0, Duplex, Speed)) {  // Notify interface
 #0 about the current settings. Return value indicates if there was a change from the last settings.
    _UpdateMACSettings(Duplex, Speed);
  // Reconfigure the MAC if the settings are confirmed to be different.
  }
}

/*********************************************************************
*
*       _SendPacket()
*
*  Function description
*    Sends the next packet in the Tx FIFO.
*    Function is called from a task via function pointer in driver
*    API table or from Tx-interrupt. Locking is done using TxIsBusy.
*/
static void _SendPacket(void) {
  void*    pPacket;   // Pointer to raw packet data to send.
  unsigned NumBytes;  // Size of raw packet data to send.

  //
  // Check if we can send more.
  //
  NumBytes = IP_GetNextOutPacketFast(&pPacket);
  if (NumBytes == 0) {  // No more packets to send ?
    _TxIsBusy = 0;      // Unlock TxIsBusy to allow sending from task.
    return;
  }
  IP_LOG((IP_MTYPE_DRIVER, "DRIVER: Sending packet: %d bytes", NumBytes));
  INC(TxSendCnt);
  //
  // Send packet.
  //
  IP_USE_PARA(pPacket);
}
```



```
/*********************************************************************
*
*       _ISR_Handler
*
*  Function description
*    This is the interrupt service routine for the NI (MAC).
*    It handles all interrupts (Rx, Tx, Error).
*/
static void _ISR_Handler(void) {
  U32                   v;

  v = 0;  // Dummy status for this sample. Interrupt without any flags should never occur.
  //
  // Read interrupt status.
  //
  v = MAC_ISR_STATUS;  // Read interrupt status.
  MAC_ISR_STATUS = v;  // Clear flags if interrupts are write-to-clear.
  //
  // Handle Rx.
  //
  if (v & MAC_ISR_STATUS_RX_MASK) {
    INC(RxIntCnt);
    IP_OnRx();
  }
  //
  // Handle Tx.
  //
  if (v & MAC_ISR_STATUS_TX_MASK) {
    INC(TxIntCnt);
    if (_TxIsBusy) {
      IP_RemoveOutPacket();  // Remove last sent packet from FIFO.
      _SendPacket();         // Try to send more.
    } else {
      IP_WARN_INTERNAL((IP_MTYPE_DRIVER, "DRIVER: Tx complete interrupt, but no packet
 sent."));
    }
  }
}

/*********************************************************************
*
*       _cbInit()
*
*  Function description
*    General initialization function of the driver.
*    Called by the stack in the initialization phase, typically
*    before any other driver function.
*    Function is called from a task via function pointer in Driver
*    API table.
*
*  Parameters
*    IFaceId   : Zero-based interface index.
*
*  Return value
*    == 0: O.K.
*    != 0: Error.
*/
static int _cbInit(unsigned IFaceId) {
  BSP_IP_INSTALL_ISR_PARA ISRPara;

  //
  // Initialize port pins.
  //
  IP_BSP_Init(IFaceId);
  //
  // Setup either MII (default) or RMII mode if MAC needs to know.
  //
  if (_PHY_Context.UseRMII != 0) {
    //
    // Set to RMII mode.
    //
  } else {
    //
    // Set to MII mode.
    //
```



```
  }
  //
  // Init PHY and update link state.
  //
  _pPHY_Driver->pfInit(&_PHY_Context);
  _UpdateLinkState();
  //
  // Set ISR routine.
  //
  IP_MEMSET(&ISRPara, 0, sizeof(ISRPara));
  ISRPara.pfISR = _ISR_Handler;
  IP_BSP_InstallISR(IFaceId, &ISRPara);
  //
  // Init and configure the MAC and start Rx & Tx.
  //
  _IsInited = 1;
  return 0;  // O.K.
}

/*********************************************************************
*
*       _cbSendPacketFromTask()
*
*  Function description
*    Sends the next packet in the Tx FIFO. Called from a task
*    via function pointer in Driver API table.
*
*  Parameters
*    IFaceId: Zero-based interface index.
*
*  Return value
*    == 0: O.K.
*    != 0: Error.
*/
static int _cbSendPacketFromTask(unsigned IFaceId) {
  IP_USE_PARA(IFaceId);

  if (_TxIsBusy == 0) {  // Allowed to send from task ?
    _TxIsBusy = 1;       // Further send operations will be done via ISR.
    _SendPacket();
  }
  return 0;  // O.K.
}

/*********************************************************************
*
*       _cbGetPacketSize()
*
*  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 API table if IP_RxTask is running or directly from
*    the Rx ISR.
*
*  Parameters
*    IFaceId: Zero-based interface index.
*
*  Return value
*    >  0: Required size for packet buffer to store the received data.
*    == 0: No further received.
*/
static int _cbGetPacketSize(unsigned IFaceId) {
  IP_USE_PARA(IFaceId);

  //
  // Check for size of next frame received and return
  // its size or 0 in case there is no next frame.
  //
  return 0;
}

/*********************************************************************
*
*       _cbReadPacket()
*
```



```
*  Function description
*    Reads one frame received into a packet buffer.
*    NumBytes is the number of bytes as retrieved by _cbGetPacketSize().
*    Function is called from a task via function pointer in Driver
*    API table.
*
*  Parameters
*    IFaceId : Zero-based interface index.
*    pDest   : Pointer to buffer where to copy the packet content. Can
*              be NULL which means discard data.
*    NumBytes: Number of bytes to copy from received packet.
*
*  Return value
*    == 0: O.K.
*    != 0: Error.
*/
static int _cbReadPacket(unsigned IFaceId, U8* pDest, unsigned NumBytes) {
  IP_USE_PARA(IFaceId);

  INC(RxCnt);
  if (pDest != NULL) {  // Copy data into packet buffer ?
    IP_LOG((IP_MTYPE_DRIVER, "Packet: %d Bytes --- Read.", NumBytes));
  } else {              // No free packet buffer, discard data.
    IP_LOG((IP_MTYPE_DRIVER, "Packet: %d Bytes --- Discarded.", NumBytes));
  }
  return 0;  // O.K.
}

/*********************************************************************
*
*       _cbTimer()
*
*  Function description
*    Timer function called by the IP_Task once per second.
*    Function is called from a task via function pointer in Driver
*    API table.
*
*  Parameters
*    IFaceId: Zero-based interface index.
*/
static void _cbTimer(unsigned IFaceId) {
  IP_USE_PARA(IFaceId);

  _UpdateLinkState();
}

/*********************************************************************
*
*       _GetCaps()
*
*  Function description
*    Fills a structure with information about the capabilities
*    like number of supported precise filters, support of
*    hash filtering, support of promiscuous mode.
*
*  Parameters
*    pCaps: Pointer to an element of type IP_NI_CMD_GET_CAPS_EX_DATA .
*
*  Return value
*    == 0: O.K.
*/
static int _GetCaps (IP_NI_CMD_GET_CAPS_EX_DATA* pCaps) {
//  pCaps->NumPreciseFilter   = 1;  // Number of supported precise filters.
//  pCaps->HasHashFilter      = 1;  // Hash filter supported?       0: Not supported,
 1: Supported.
//  pCaps->HasPromiscuousMode = 1;  // Promiscuous mode supported?  0: Not supported,
 1: Supported.
//  pCaps->PacketDataShiftCnt = 0;
  // ShiftCnt for ideal packet data alignment. Could avoid a memcpy into an extra buffer that is aligned.
  IP_USE_PARA(pCaps);
  return 0;
}

/*********************************************************************
*
*       _cbControl()
```



```
*
*  Function description
*    This routine allows extending the function table dynamically for
*    several operations.
*
*  Parameters
*    IFaceId: Zero-based interface index.
*    Cmd    : Command to execute. Most common commands are:
*               * IP_NI_CMD_SET_FILTER
*               * IP_NI_CMD_GET_CAPS
*               * IP_NI_CMD_SET_PHY_ADDR
*               * IP_NI_CMD_SET_PHY_MODE
*               * IP_NI_CMD_POLL
*               * IP_NI_CMD_SET_SUPPORTED_DUPLEX_MODES
*               * IP_NI_CMD_GET_CAPS_EX
*    p      : Pointer to data processed by command.
*
*  Return value
*    >=  0: O.K.
*    == -1: Error or command not supported.
*/
static int _cbControl(unsigned IFaceId, int Cmd, void* p) {
  int r;

  IP_USE_PARA(IFaceId);

  r = -1;  // Assume command is not supported.

  switch (Cmd) {
  case IP_NI_CMD_SET_FILTER:
    r = _SetFilter((IP_NI_CMD_SET_FILTER_DATA*)p);
    break;
  case IP_NI_CMD_GET_CAPS:
    r = 0
//        | IP_NI_CAPS_WRITE_IP_CHKSUM
  // Driver capable of inserting the IP-checksum into an outgoing packet ?
//        | IP_NI_CAPS_WRITE_UDP_CHKSUM
  // Driver capable of inserting the UDP-checksum into an outgoing packet ?
//        | IP_NI_CAPS_WRITE_TCP_CHKSUM
  // Driver capable of inserting the TCP-checksum into an outgoing packet ?
//        | IP_NI_CAPS_WRITE_ICMP_CHKSUM
  // Driver capable of inserting the ICMP-checksum into an outgoing packet ?
//        | IP_NI_CAPS_CHECK_IP_CHKSUM
  // Driver capable of computing and comparing the IP-checksum of an incoming packet ?
//        | IP_NI_CAPS_CHECK_UDP_CHKSUM
  // Driver capable of computing and comparing the UDP-checksum of an incoming packet ?
//        | IP_NI_CAPS_CHECK_TCP_CHKSUM
  // Driver capable of computing and comparing the TCP-checksum of an incoming packet ?
//        | IP_NI_CAPS_CHECK_ICMP_CHKSUM
  // Driver capable of computing and comparing the ICMP-checksum of an incoming packet ?
//        | IP_NI_CAPS_WRITE_ICMPV6_CHKSUM
  // Driver capable of inserting the ICMP-checksum into an outgoing packet ?
        ;
    break;
  case IP_NI_CMD_SET_PHY_ADDR:
    if (_IsInited != 0) {
      break;
    }
    _PHY_Context.Addr = (U8)(int)p;
    r = 0;  // O.K.
    break;
  case IP_NI_CMD_SET_PHY_MODE:
    if (_IsInited != 0) {
      break;
    }
    _PHY_Context.UseRMII = (U8)(int)p;
    r = 0;  // O.K.
    break;
  case IP_NI_CMD_POLL:
    //
    //  Poll MAC (typically once per ms) in cases where MAC does not trigger an interrupt.
    //
    _ISR_Handler();
    break;
  case IP_NI_CMD_SET_SUPPORTED_DUPLEX_MODES:
    if (_IsInited != 0) {
```



```
      break;
    }
    _PHY_Context.SupportedModes = (U16)(int)p;
    r = 0;  // O.K.
    break;
#if IP_ALLOW_DEINIT
  case IP_NI_CMD_DEINIT:
    //
    // Stop Rx & Tx.
    // Make sure that the Rx & Tx interrupts are disabled.
    // De-initialize the MAC.
    //
    IP_BSP_DeInit(IFaceId);
    r = 0;  // O.K.
    break;
#endif
  case IP_NI_CMD_GET_CAPS_EX:
    r = _GetCaps((IP_NI_CMD_GET_CAPS_EX_DATA*)p);
    break;
  }
  return r;
}

/*********************************************************************
*
*       Global functions
*
**********************************************************************
*/

//None

/*********************************************************************
*
*       Driver API Table
*
**********************************************************************
*/
const IP_HW_DRIVER IP_Driver_Template = {
  _cbInit,                // pfInit
  _cbSendPacketFromTask,  // pfSendPacket
  _cbGetPacketSize,       // pfGetPacketSize
  _cbReadPacket,          // pfReadPacket
  _cbTimer,               // pfTimer
  _cbControl,             // pfControl
  NULL                    // pfEnDisableRxInt
};

/************************** End of file ****************************/
```


Chapter 20

PHY drivers

emNet has been designed to cooperate with any kind of hardware. Typically almost any PHY is compatible with the emNet 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.

20.1 PHY drivers general information

To use emNet 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.

20.1.1 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.

20.2 Available PHY drivers

emNet 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 emNet.

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

PHY driver	Identifier
Generic driver	IP_PHY_Driver_Generic
Micrel Switch PHY driver	IP_PHY_Driver_Micrel_Switch_<Product Name>
Marvell 88E1111 Fiber PHY driver	IP_PHY_Driver_MARVELL_88E1111_Fiber

To add a PHY driver to emNet, IP_PHY_AddDriver() should be called from within IP_X_Config() with the proper identifier. Refer to *IP_PHY_AddDriver* on page 189 for detailed information.

20.2.1 Generic driver

The emNet 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 emNet 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	0K Bytes

All required RAM is taken from the RAM that has been assigned to emNet using IP_AddMemory(). Only a few bytes are required.

20.2.1.1 Generic PHY driver API functions

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

Function	Description
IP_PHY_GENERIC_RemapAccess()	This function allows remapping the access routines of a PHY interface.

20.2.1.2 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);
```

Parameters

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

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.

20.2.2 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 emNet 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[_HostPort]`
KSZ8863	`IP_PHY_Driver_Micrel_Switch_KSZ8863[_HostPort]`
KSZ8895	`IP_PHY_Driver_Micrel_Switch_KSZ8895[_HostPort]`

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	0K Bytes

All required RAM is taken from the RAM that has been assigned to emNet using `IP_AddMemory()`. Only a few bytes are required.

20.2.2.1 Micrel Switch PHY driver API functions

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

Function	Description
`IP_PHY_MICREL_SWITCH_AssignPortNumber()`	This function assigns the physical switch port number to an interface.
`IP_PHY_MICREL_SWITCH_ConfigLearnDisable()`	This function can set the learn disable config for a specific port to enabled/disabled.
`IP_PHY_MICREL_SWITCH_ConfigRxEnable()`	This function can set Rx (network to switch) for a specific port to enabled/disabled.
`IP_PHY_MICREL_SWITCH_ConfigTailTagging()`	This function switches Tail Tagging on/off.
`IP_PHY_MICREL_SWITCH_ConfigTxEnable()`	This function can set Tx (switch to network) for a specific port to enabled/disabled.
`IP_PHY_MICREL_SWITCH_ConfigUseInternalRmiiClock()`	This function selects the clock source for the RMII host port.

20.2.2.2 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);
```

Parameters

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

20.2.2.3 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);
```

Parameters

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

20.2.2.4 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);
```

Parameters

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

20.2.2.5 IP_PHY_MICREL_SWITCH_ConfigTailTagging()

Description

This function switches Tail Tagging on/off.

Prototype

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

Parameters

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

Additional information

Needs to be used with the interface of the host port of the switch.

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.

20.2.2.6 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);
```

Parameters

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

20.2.2.7 IP_PHY_MICREL_SWITCH_ConfigUseInternalRmiiClock()

Description

This function selects the clock source for the RMII host port. The default is to use the externally provided clock.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
OnOff	• 0: Use external clock. • 1: Use internal clock.

Additional information

This setting is only relevant for parts that allow a clock selection. The correct clock selection depends on the hardware layout. Details can be found in the datasheet of the switch e.g. in a chapter called “RMII INTERFACE OPERATION” for the KSZ8863 .

20.2.3 Marvell 88E1111 Fiber PHY driver

Although the the Marvell 88E1111 PHY in copper mode is supported out of the box by the *Generic driver* on page 593, this PHY supports a Fiber mode as well. The Fiber mode requires a different handling and therefore requires a separate driver to handle this mode.

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 0.8 kBytes	0K Bytes

All required RAM is taken from the RAM that has been assigned to emNet using `IP_AddMemory()`. Only a few bytes are required.

Chapter 21

WiFi drivers

emNet allows to easily add WiFi (`IEEE 802.11`) support to your project. Using one of the supported WiFi modules and available driver for it allows a WiFi interface to be added in nearly the same easy way as adding any other emNet interface.

As almost each WiFi module uses a different API and a more or less complete documentation of it, integration can become very time consuming. The emNet WiFi drivers provide an unified API to different WiFi modules that allows an easy integration into your project in short time.

This chapter lists drivers that require special configuration or come with their own extended API that allows a configuration beyond the regular emNet API.

21.1 WiFi drivers general information

There are two different types of WiFi drivers used with emNet:

- Network Interface WiFi drivers
- WiFi PHY bridges

Both types make use of the same API for your application but a re different in their how they provide WiFi support in terms of their hardware.

21.1.1 Network Interface WiFi drivers

The typical form of an emNet WiFi driver is used in the same way as a regular Network Interface (NI) driver. It basically works like an external Ethernet controller, typically using communication via a peripheral interface like SPI. Other common interfaces used are I2C, UART or SDIO.

To the stack itself this is just another interface that is added like it would be for an integrated or external Ethernet controller.

Using a free peripheral interface this solution can not only be used to build a new hardware but can be used to upgrade existing solutions with WiFi as well.

While providing WiFi support by the simple usage of a peripheral interface like SPI, often an SDK of the WiFi module vendor is required for communication with the module using vendor specific commands. In some cases using the WiFi modules of a vendor is even prohibited without the vendor SDK.

This type of driver is added to the stack like a regular driver by calling `IP_WIFI_AddInterface()`.

21.1.2 WiFi PHY bridges

Some WiFi solutions offer to convert a regular Ethernet device into a WiFi solution by providing a bridge device. For a regular cable based setup a copper cable PHY would be connected to the Ethernet MAC of the MCU (or an external MAC). Instead of the copper cable PHY a WiFi bridge can be connected, replacing the PHY. These devices are providing a so-called MAC(of the MCU)-to-MAC(of the WiFi module) mode and are interfaced via (R)MII the same way as a regular PHY would be.

Ethernet communication takes place using (R)MII and therefore requires an Ethernet MAC to be present. A second configuration interface like SPI or UART is required as well as (R)MII is used for Ethernet data only.

Using this solution an existing design can be converted to a WiFi solution by basically exchanging the PHY. The separation of the Ethernet data and configuration has the chance to reach higher transfer speeds than a single unidirectional interface handling Ethernet and configuration at the same time.

Like any other PHY driver an Ethernet controller and its driver is still required. This means that first an Ethernet driver has to be added calling `IP_AddEtherInterface()` and then the driver of the WiFi PHY bridge has to be added and assigned to this interface by calling `IP_PHY_AddDriver()`.

21.2 List of special WiFi drivers

Several WiFi drivers are available for emNet to provide an unified emNet WiFi API to the vendor specific API. The vendor specific API is different between vendors and might even be different for modules from the same vendor. In general all drivers are added to the stack in the same way:

To add a WiFi driver to emNet, `IP_WIFI_AddInterface()` should be called from within `IP_X_Config()` with the proper identifier.

To add a WiFi PHY bridge driver to emNet, `IP_PHY_AddDriver()` on page 132 should be called from within `IP_X_Config()` with the proper identifier.

Depending on the module and driver they might require additional configuration or provide additional features. These additional configuration functions are listed here. The following WiFi drivers are described in detail with their API:

WiFi drivers with special functions
ConnectOne IW
Redpine Signals RS9113

21.2.1 ConnectOne IW

The emNet ConnectOne IW driver supports all WiFi<->LAN bridges sharing the same AT +i command set which basically are all WiFi modules named "iW-*" using the CO2144 or the older CO2128 core CPU.

21.2.1.1 Hardware access abstraction

The ConnectOne IW driver requires a hardware access API to be passed to `IP_PHY_AddDriver()` to send/receive configuration data via SPI or UART.

A sample for a SPI connection is shipped with the driver as reference implementation.

```
typedef struct {
  void (*pfHWReset)      (unsigned IFaceId);
  int  (*pfSendATCommand)(unsigned IFaceId, U32 Timeout, const char* sCmd);
  int  (*pfClrBuf)       (unsigned IFaceId, U32 Timeout, char IgnoreSpiInt);
  int  (*pfLoadLine)     (unsigned IFaceId, U32 Timeout);
  int  (*pfReadLine)     (unsigned IFaceId, U32 Timeout, char* pBuffer,
unsigned BufferSize);
} IP_PHY_WIFI_CONNECTONE_IW_ACCESS;
```

Member	Description
`pfHWReset`	Reset the WiFi module. Shall only return after the module reset has been successfully asserted and deasserted. Only used in case of an error. The module shall be manually hardware reset once during startup.
`pfSendATCommand`	Sends the given AT+i command to the module. CR&LF are included in the string to send.
`pfClrBuf`	Reads data from the module until `SPI_INT` gets deasserted. If `IgnoreSpiInt` is set the module could be in an unknown state after an error and the `SPI_INT` line might no longer be reliable. In this case just try to read data for some time to clear all data left in the module for a fresh start.
`pfLoadLine`	Reads in data in chunks as sent by the WiFi module until the buffer is full or a complete line is in the buffer. A static buffer is expected to be used to first assemble the chunks into a complete message.
`pfReadLine`	Checks if a complete line is in static buffer and copies it to the provided buffer.

21.2.1.2 ConnectOne IW driver API functions

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

Function	Description
`IP_PHY_WIFI_CONNECTONE_IW_ConfigSPI()`	Sets the `SPI_INT` port the module uses to signal an event.

21.2.1.3 IP_PHY_WIFI_CONNECTONE_IW_ConfigSPI()

Description

Sets the SPI_INT port the module uses to signal an event.

Prototype

```
void IP_PHY_WIFI_CONNECTONE_IW_ConfigSPI(unsigned IFaceId,
                                         char     SPIConfig);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
SPIConfig	SPI_INT port configuration as mentioned in the AT+i command documentation.

21.2.2 Redpine Signals RS9113

The emNet Redpine Signals RS9113 driver supports the RS9113 Connect-io-n and WiseConnect family and modules compatible.

21.2.2.1 Redpine Signals RS9113 driver API functions

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

Function	Description
IP_NI_WIFI_REDPINE_RS9113_ConfigAntenna()	Selects the internal or external antenna and configures the gain for 2.4GHz and 5GHz modes.
IP_NI_WIFI_REDPINE_RS9113_ConfigRegion()	Selects a world region and allowed channel configuration.
IP_NI_WIFI_REDPINE_RS9113_SetAccessPointParameters()	Configures access point parameters.
IP_NI_WIFI_REDPINE_RS9113_SetSpiSpeedChangeCallback()	Sets a callback to be executed when the SPI interface can be used in high speed mode (above 25MHz) or when it is reset due to a module reset to recover from an error.
IP_NI_WIFI_REDPINE_RS9113_SetUpdateCallback()	Sets a callback to be executed during boot that updates the module firmware.

21.2.2.2 IP_NI_WIFI_REDPINE_RS9113_ConfigAntenna()

Description

Selects the internal or external antenna and configures the gain for 2.4GHz and 5GHz modes.

Prototype

```
int IP_NI_WIFI_REDPINE_RS9113_ConfigAntenna(unsigned IFaceId,
                                            U8       Antenna,
                                            U8       Gain_24GHz,
                                            U8       Gain_5GHz);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Antenna	• 0: Internal antenna • 1: External antenna
Gain_24GHz	0..10 (according to RS9113 API documentation)
Gain_5GHz	0..10 (according to RS9113 API documentation)

Return value

= 0 O.K.
≠ 0 Error.

Additional information

The value that can be configured for the antenna gain is not explained in detail in the RS9113 API documentation. The values are passed to the module without modification.

21.2.2.3 IP_NI_WIFI_REDPINE_RS9113_ConfigRegion()

Description

Selects a world region and allowed channel configuration.

Prototype

```
int IP_NI_WIFI_REDPINE_RS9113_ConfigRegion(unsigned IFaceId,
                                           U8       Region);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Region	• IP_NI_WIFI_REDPINE_RS9113_REGION_AUTO_DETECT • IP_NI_WIFI_REDPINE_RS9113_REGION_US • IP_NI_WIFI_REDPINE_RS9113_REGION_EUROPE • IP_NI_WIFI_REDPINE_RS9113_REGION_JAPAN • IP_NI_WIFI_REDPINE_RS9113_REGION_WORLD

Return value

= 0 O.K.
≠ 0 Error.

Additional information

Has to be called during init.

For a list of channels that are enabled for each region, please refer to the RS9113 API documentation and search for "Region_code".

21.2.2.4 IP_NI_WIFI_REDPINE_RS9113_SetAccessPointParameters()

Description

Configures access point parameters. If not called, default values are used.

Prototype

```
void IP_NI_WIFI_REDPINE_RS9113_SetAccessPointParameters
                                    (      unsigned                       IFaceId,
                                     const IP_NI_WIFI_RS9113_AP_CONFIG * pParams);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pParams	Pointer to the parameters structure.

Additional information

For details on the parameters value refer to the RS9113 API documentation of the function `rsi_set_ap_config()`.

21.2.2.5 IP_NI_WIFI_REDPINE_RS9113_SetSpiSpeedChangeCallback()

Description

Sets a callback to be executed when the SPI interface can be used in high speed mode (above 25MHz) or when it is reset due to a module reset to recover from an error.

Prototype

```
void IP_NI_WIFI_REDPINE_RS9113_SetSpiSpeedChangeCallback
                            (unsigned                                    IFaceId,
                             IP_NI_WIFI_REDPINE_RS9113_pfOnSpiSpeedChange pf);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pf	Callback to execute on a change.

21.2.2.6 IP_NI_WIFI_REDPINE_RS9113_SetUpdateCallback()

Description

Sets a callback to be executed during boot that updates the module firmware.

Prototype

```
void IP_NI_WIFI_REDPINE_RS9113_SetUpdateCallback
                                  (unsigned                                  IFaceId,
                                   IP_NI_WIFI_REDPINE_RS9113_pfUpdateModule pf);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pf	Callback to execute the update.

Chapter 22

Configuring emNet

emNet 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 emNet 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.

22.1 Runtime configuration

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

22.1.1 IP_X_Config()

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 165 for more information.

Example

```
/*********************************************************************
*               SEGGER MICROCONTROLLER SYSTEME GmbH                  *
*        Solutions for real time microcontroller applications        *
**********************************************************************
*                                                                    *
*        (c) 2007         SEGGER Microcontroller Systeme GmbH        *
*                                                                    *
*        Internet: www.segger.com    Support:  support@segger.com    *
*                                                                    *
**********************************************************************
----------------------------------------------------------------------
File        : IP_Config_Template_ETH.c
Purpose     : Configuration file template
---------------------------END-OF-HEADER------------------------------
*/

#include "IP.h"
#include "IP_NI_TEMPLATE.h"
/*********************************************************************
*
*       Configuration
*
**********************************************************************
*/

#define ALLOC_SIZE   0x6000
  // Size of memory dedicated to the stack in bytes.
#define DRIVER       &IP_Driver_Template         // Driver used for target.
#define TARGET_NAME  "TARGET"                    // Target name used for DHCP client.
#define HW_ADDR      "\x00\x22\xC7\xFF\xFF\xFF"  // MAC addr. used for target.
#define USE_DHCP     1
  // Use DHCP client or static IP configuration.

//
// The following parameters are only used when the DHCP client is not active.
//
#define IP_ADDR      IP_BYTES2ADDR(192, 168,   2, 252)
#define SUBNET_MASK  IP_BYTES2ADDR(255, 255, 255,   0)
#define GW_ADDR      IP_BYTES2ADDR(192, 168,   2,   1)
#define DNS_ADDR     IP_BYTES2ADDR(192, 168,   2,   1)

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/

static U32 _aPool[ALLOC_SIZE / 4];
  // This is the memory area used by the stack.

/*********************************************************************
*
*       Global functions
*
```



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

/*********************************************************************
*
*       IP_X_Config()
*
*  Function description
*    This function is called by the IP stack during IP_Init().
*
*  Notes
*    Typical memory/buffer configurations:
*      Microcontroller system, minimum size optimized
*        #define ALLOC_SIZE 0x1000                          // 4KBytes RAM.
*        mtu = 576;                                         // 576 is minimum acc. to RFC,
 1500 is max. for Ethernet.
*        IP_SetMTU(0, mtu);                                 // Maximum Transmission Unit is
 1500 for Ethernet by default.
*        IP_AddBuffers(4, 256);                             // Small buffers.
*        IP_AddBuffers(2, mtu + 16);                        // Big buffers. Size should be
 mtu + 16 byte for Ethernet header (2 bytes type, 2*6 bytes MAC, 2 bytes padding).
*        IP_ConfTCPSpace(2 * (mtu - 40), 1 * (mtu - 40));  // Define the TCP Tx and Rx
 window size. At least Tx space for 2*(mtu-40) for two full TCP packets is needed.
*
*      Microcontroller system, size optimized
*        #define ALLOC_SIZE 0x3000                          // 12KBytes RAM.
*        mtu = 576;                                         // 576 is minimum acc. to RFC,
 1500 is max. for Ethernet.
*        IP_SetMTU(0, mtu);                                 // Maximum Transmission Unit is
 1500 for Ethernet by default.
*        IP_AddBuffers(8, 256);                             // Small buffers.
*        IP_AddBuffers(4, mtu + 16);                        // Big buffers. Size should be
 mtu + 16 byte for Ethernet header (2 bytes type, 2*6 bytes MAC, 2 bytes padding).
*        IP_ConfTCPSpace(2 * (mtu - 40), 2 * (mtu - 40));  // Define the TCP Tx and Rx
 window size. At least Tx space for 2*(mtu-40) for two full TCP packets is needed.
*
*      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 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. At least Tx space for 2*(mtu-40) for two full TCP packets is needed.
*
*      System with lots of RAM
*        #define ALLOC_SIZE 0x20000                         // 128 KBytes RAM.
*        mtu = 1500;                                        // 576 is minimum acc. to RFC,
 1500 is max. for Ethernet.
*        IP_SetMTU(0, mtu);                                 // Maximum Transmission Unit is
 1500 for Ethernet by default.
*        IP_AddBuffers(50, 256);                            // Small buffers.
*        IP_AddBuffers(50, mtu + 16);                       // Big buffers. Size should be
 mtu + 16 byte for Ethernet header (2 bytes type, 2*6 bytes MAC, 2 bytes padding).
*        IP_ConfTCPSpace(8 * (mtu - 40), 8 * (mtu - 40));  // Define the TCP Tx and Rx
 window size. At least Tx space for 2*(mtu-40) for two full TCP packets is needed.
*/
void IP_X_Config(void) {
  int mtu;
  int IFaceId;

  IP_AssignMemory(_aPool, sizeof(_aPool));
  // Assigning memory should be the first thing.
  IFaceId = IP_AddEtherInterface(DRIVER);          // Add driver for your hardware.
  IP_SetHWAddrEx(IFaceId, (const U8*)HW_ADDR, 6);
  // MAC addr.: Needs to be unique for production units.
  //
  // Configure the PHY interface mode (optional):
  //   - IP_PHY_MODE_MII : MII, typically default if not explicitly configured.
  //   - IP_PHY_MODE_RMII: RMII
  // Can be set/overwritten from BSP_IP pfGetMiiMode() callback.
  //
//  IP_NI_ConfigPHYMode(IFaceId, IP_PHY_MODE_RMII);
```



```
  //
  // Use DHCP client or define IP address, subnet mask,
  // gateway address and DNS server according to the
  // requirements of your application.
  //
#if USE_DHCP
  IP_DHCPC_Activate(IFaceId, TARGET_NAME, NULL, NULL);  // Activate DHCP client.
#else
  IP_SetAddrMaskEx(IFaceId, IP_ADDR, SUBNET_MASK);      // Assign IP addr. and subnet mask.
  IP_SetGWAddr(IFaceId, GW_ADDR);                       // Set gateway addr.
  IP_DNS_SetServer(DNS_ADDR);                           // Set DNS server addr.
#endif
  //
  // 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(IFaceId, 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 the TCP Tx and Rx window size
  IP_SOCKET_SetDefaultOptions(0
//                              | SO_TIMESTAMP
  // Send TCP timestamp to optimize the round trip time measurement. Normally not used in LAN.
                              | SO_KEEPALIVE
  // Enable keepalives by default for TCP sockets.
                             );
  //
  // 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
                  | IP_MTYPE_APPLICATION     // Output application messages.
                  | IP_MTYPE_INIT            // Output all messages from init.
                  | IP_MTYPE_LINK_CHANGE     // Output a message if link status changes.
                  | IP_MTYPE_PPP             // Output all PPP/PPPoE related messages.
                  | IP_MTYPE_DHCP            // Output general DHCP status messages.
#if IP_SUPPORT_IPV6
                  | IP_MTYPE_IPV6            // Output IPv6 address related messages
#endif
//                  | IP_MTYPE_DHCP_EXT        // Output additional DHCP messages.
//                  | IP_MTYPE_CORE            // Output log messages from core module.
//                  | IP_MTYPE_ALLOC           // Output log messages for memory allocation.
//                  | IP_MTYPE_DRIVER          // Output log messages from driver.
//                  | IP_MTYPE_ARP             // Output log messages from ARP layer.
//                  | IP_MTYPE_IP              // Output log messages from IP layer.
//                  | IP_MTYPE_TCP_CLOSE
  // Output a log messages if a TCP connection has been closed.
//                  | IP_MTYPE_TCP_OPEN
  // Output a log messages if a TCP connection has been opened.
//                  | IP_MTYPE_TCP_IN          // Output TCP input logs.
//                  | IP_MTYPE_TCP_OUT         // Output TCP output logs.
//                  | IP_MTYPE_TCP_RTT         // Output TCP round trip time (RTT) logs.
//                  | IP_MTYPE_TCP_RXWIN       // Output TCP RX window related log messages.
//                  | IP_MTYPE_TCP             // Output all TCP related log messages.
//                  | IP_MTYPE_UDP_IN          // Output UDP input logs.
//                  | IP_MTYPE_UDP_OUT         // Output UDP output logs.
//                  | IP_MTYPE_UDP             // Output all UDP related messages.
//                  | IP_MTYPE_ICMP            // Output ICMP related log messages.
//                  | IP_MTYPE_NET_IN          // Output network input related messages.
//                  | IP_MTYPE_NET_OUT         // Output network output related messages.
//                  | IP_MTYPE_DNS             // Output all DNS related messages.
//                  | IP_MTYPE_SOCKET_STATE    // Output socket status messages.
//                  | IP_MTYPE_SOCKET_READ     // Output socket read related messages.
//                  | IP_MTYPE_SOCKET_WRITE    // Output socket write related messages.
//                  | IP_MTYPE_SOCKET          // Output all socket related messages.
                );
  //
```



```
  // Add protocols to the stack.
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
#if IP_SUPPORT_IPV6
  IP_IPV6_Add(IFaceId);
#endif
}

/************************** End of file ***************************/
```


22.1.2 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.

22.1.3 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

22.1.3.1 RAM for TCP window

The data for the TCP window is stored in packet buffers. The number of packet buffers required for best performance per socket is typically:

(WindowSize / PacketBufferSize)

This amount of buffers (and RAM for these buffers) is needed for every simultaneously active TCP connection, where “active” means sending & receiving data.

If a connection is used bidirectional for sending & receiving data at the same time, enough buffers should be added to be able to support the full TxWindowsSize and RxWindowSize at the same time.

> **Note**
>
> The configuration mentioned above is not necessary if not all of your connections are using their maximum WindowSize all the time to reach the best speed possible.
>
> The TCP protocol can freely assign free packet buffers in the system to all socket buffers. Therefore it is not necessary to reserve the maximum number of packet buffers for all sockets that could be used at the same time if you can afford retransmits and their delays of typically ~200ms.
>
> On a shortage, especially when failing to get a buffer for adding received data to a socket buffer (data gets discarded in the TCP layer), the TCP protocol is able to cope with this situation by not ACKing the data and having the peer retransmit it again, giving the application a chance to free some buffers in the meantime.

22.1.3.2 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.
                                             // to RFC, 1500 is max. for Ethernet
IP_SetMTU(0, mtu);                           // Maximum Transmission Unit is 1500
```



```
                                          // for Ethernet by default
IP_AddBuffers(4, 256);                    // Small buffers.
IP_AddBuffers(2, mtu + 16);               // Big buffers. Size should be mtu
                                          // + 16 byte for Ethernet header
                                          // (2 bytes type, 2*6 bytes MAC,
                                          // 2 bytes padding)
IP_ConfTCPSpace(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.
                                          // to RFC, 1500 is max. for Ethernet
IP_SetMTU(0, mtu);                        // Maximum Transmission Unit is 1500
                                          // for Ethernet by default
IP_AddBuffers(8, 256);                    // Small buffers.
IP_AddBuffers(4, mtu + 16);               // Big buffers. Size should be mtu
                                          // + 16 byte for Ethernet header
                                          // (2 bytes type, 2*6 bytes MAC,
                                          // 2 bytes padding)
IP_ConfTCPSpace(2 * (mtu-40), 2 * (mtu-40));  // Define TCP Tx and Rx window size
```

Example configuration - Normal (24 kBytes)

This configuration is a typical configuration for many MCUs that have a fair amount of internal RAM. It can be used for applications which are designed for a higher amount of traffic and/or multiple client connections.

```
#define ALLOC_SIZE 0x6000                 // 24 kBytes RAM
mtu = 1500;                               // 576 is minimum acc. to RFC,
                                          // 500 is max. for Ethernet
IP_SetMTU(0, mtu);                        // Maximum Transmission Unit is 1500
                                          // for Ethernet by default
IP_AddBuffers(12, 256);                   // Small buffers.
IP_AddBuffers(6, mtu + 16);               // Big buffers. Size should be mtu
                                          // + 16 byte for Ethernet header
                                          // (2 bytes type, 2*6 bytes MAC,
                                          // 2 bytes padding)
IP_ConfTCPSpace(3 * (mtu-40), 3 * (mtu-40));  // Define TCP Tx and Rx window size
```

Example configuration - Large (128 kBytes)

This configuration is a large configuration intended to be used on MCUs with many external RAM. It can be used for applications which are designed for a high amount of traffic and multiple client/server connections at the same time.

```
#define ALLOC_SIZE 0x20000                // 128 kBytes RAM
mtu = 1500;                               // 576 is minimum acc. to RFC,
                                          // 1500 is max. for Ethernet
IP_SetMTU(0, mtu);                        // Maximum Transmission Unit is 1500
                                          // for Ethernet by default
IP_AddBuffers(50, 256);                   // Small buffers.
IP_AddBuffers(50, mtu + 16);              // Big buffers. Size should be mtu
                                          // + 16 byte for Ethernet header
                                          // (2 bytes type, 2*6 bytes MAC,
                                          // 2 bytes padding)
IP_ConfTCPSpace(8 * (mtu-40), 8 * (mtu-40));  // Define TCP Tx and Rx window size
```

Warning

Do not configure "1 * (mtu-40)" (only one one packet) for the TxWindowSize if TCP delayed ACKs are enabled (default). This might lead to a delay of ~200ms before the peer ACKs the sent data as it will wait for its retransmit delay time (default ~200ms) for at least a second packet to arrive.

If you intend to run a super small configuration with only one packet for the TxWindowSize, please disable delayed ACK support in the stack by configuring the define `IP_SUPPORT_TCP_DELAYED_ACK` to `0`.

22.2 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).

22.2.1 Compile-time configuration switches

Type	Symbolic name	Default	Description
System configuration macros			
B	IP_IS_BIGENDIAN	--	Macro to define if a big endian target is used.
Statistics configuration macros			
B	IP_SUPPORT_STATS	0	Macro used as default value for all IP_SUPPORT_STATS_* defines. Leave this to 0 if you want to enable only specific stats defines.
B	IP_SUPPORT_STATS_IFACE	IP_SUPPORT_STATS	Macro to define if the emNet interface statistics should be available.
Debug macros			
N	IP_DEBUG	0	Macro to define the debug level of the emNet build. Refer to *Debug level* on page 625 for a description of the different debug level.
B	IP_SUPPORT_PROFILE	0	Macro to define if the emNet API profiling support for SystemView is used. For more information regarding SystemView please refer to https://www.segger.com/system-view.html.
B	IP_SUPPORT_PROFILE_END_CALL	0	Macro to define if the emNet API profiling support for SystemView recognizes the exact end of functions as well. For more information regarding SystemView please refer to https://www.segger.com/system-view.html.

Type	Symbolic name	Default	Description
B	IP_SUPPORT_PROFILE_FIFO	0	Macro to define if the emNet profiling support for SystemView recognizes stack internal FIFO operations as well. For more information regarding SystemView please refer to https://www.segger.com/system-view.html.
B	IP_SUPPORT_PROFILE_PACKET	0	Macro to define if the emNet profiling support for SystemView recognizes stack internal packet buffer alloc/free operations as well. For more information regarding SystemView please refer to https://www.segger.com/system-view.html.
PHY configuration macros			
N	IP_PHY_AFTER_RESET_DELAY	3	Delay [ms] between (soft) resetting the PHY and further communication with it.
IP protocol configuration macros			
B	IP_SUPPORT_IPV4	1	Enables support for IPv4. Disabling IPv4 support removes many IPv4 exclusive code passages. This can be used for example if you have enabled IPv6 and do not need/want IPv4 at all.
B	IP_SUPPORT_IPV6	0	Enables support for IPv6. Requires the IPv6 add-on.
TCP protocol configuration macros			
B	IP_SUPPORT_TCP_DELAYED_ACK	1	Enables support for TCP delayed ACKs.
N	IP_TCP_RETRANS_MIN	210	Minimum time [ms] between retransmits for segments that have not been ACKed. The real retransmit time is calculated from the round-trip-time but will stay between the min./max. values.
N	IP_TCP_RETRANS_MAX	3000	Maximum time [ms] between retransmits for segments that have not been ACKed. The real retransmit time is calculated from the round-trip-time but will stay between the min./max. values.
N	IP_TCP_RETRANS_NUM	6	Number of retransmits for segments that have not been ACKed. The retransmit time is calculated from the round-trip-time but limited to the min./max. values.
N	IP_TCP_KEEPALIVE_MAX_REPS	8	Maximum repeats of keepalive probes before dropping the connection. Keepalives need to be enabled on the socket with SO_KEEPALIVE (default).

Type	Symbolic name	Default	Description
N	IP_TCP_KEEPALIVE_INIT	10000	Timeout [ms] during the three-way-handshake connect after which keepalive probing will be started. This is not a total connect timeout. Keepalives need to be enabled on the socket with SO_KEEPALIVE (default).
N	IP_TCP_KEEPALIVE_IDLE	10000	Timeout [ms] with no segment exchange after which keepalive probing will be started. Keepalives need to be enabled on the socket with SO_KEEPALIVE (default).
N	IP_TCP_KEEPALIVE_PERIOD	10000	Time [ms] between keepalive probes sent. Keepalives need to be enabled on the socket with SO_KEEPALIVE (default).
Optimization macros			
F	IP_CKSUM	IP_cksum (C-routine in IP stack)	Macro to define an optimized checksum routine to speed up the stack. An optimized checksum routine is typically implemented in assembly language. Optimized versions for the GNU, IAR and ADS compilers are supplied.
F	IP_MEMCPY	memcpy (C-routine in standard C-library)	Macro to define an optimized memcpy routine to speed up the stack. An optimized memcpy routine is typically implemented in assembly language. Optimized version for the IAR compiler is supplied.
F	IP_MEMSET	memset (C-routine in standard C-library)	Macro to define an optimized memset routine to speed up the stack. An optimized memset routine is typically implemented in assembly language.
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.

22.2.2 Debug level

emNet can be configured to display debug information at higher debug levels to locate a problem (Error) or potential problem. To display information, emNet uses the logging routines (see chapter Debugging on page 845). These routines can be blank, they are not required for the functionality of emNet. 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 23

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

The emNet 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.

23.1 emNet IPv6

The emNet 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 emNet IPv6 add-on root directory:

Directory	Content
`Application\`	Contains the example application to test the IPv6 implementation.
`Config\`	Contains the emNet 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`

23.2 Feature list

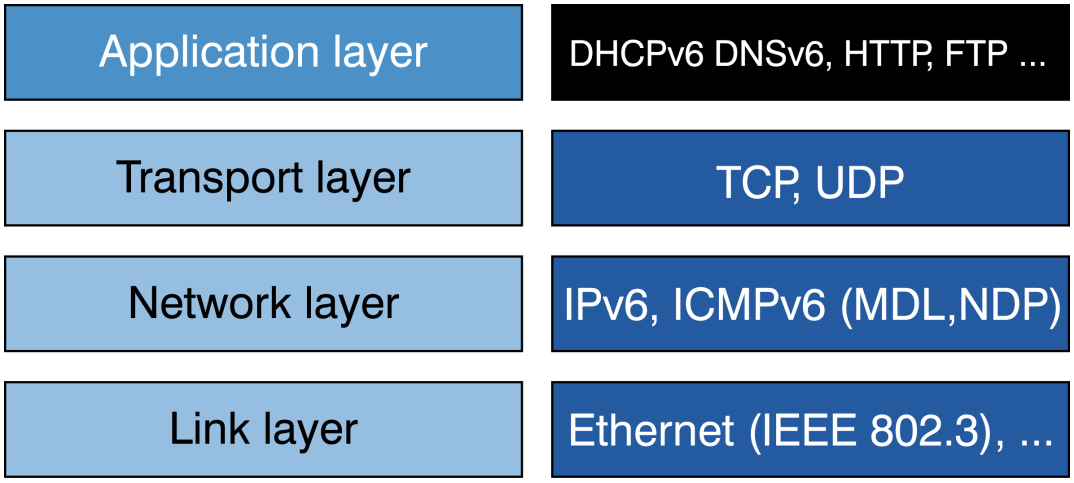
- Low memory footprint
- Easy to implement
- Internet Protocol version 6 (IPv6)
- Internet Control Message Protocol (ICMPv6)
- Neighbor Discovery Protocol (NDP)
- Multicast Listener Discovery (MLD)
- Stateless Address autoconfiguration (SLAAC)
- Standard socket interface
- No configuration required

23.3 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.



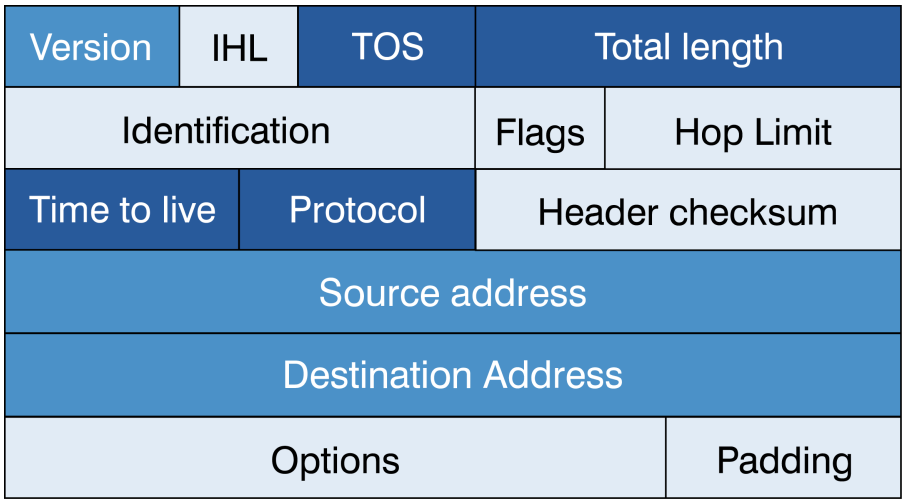
The IPv6 header has a fixed length of 40 bytes and does not include any option. Contrary to IPv4, options are stored in extension headers. The benefit of this separation is that a router never needs to parse the header so that the processing is more efficient although the header size is at least twice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

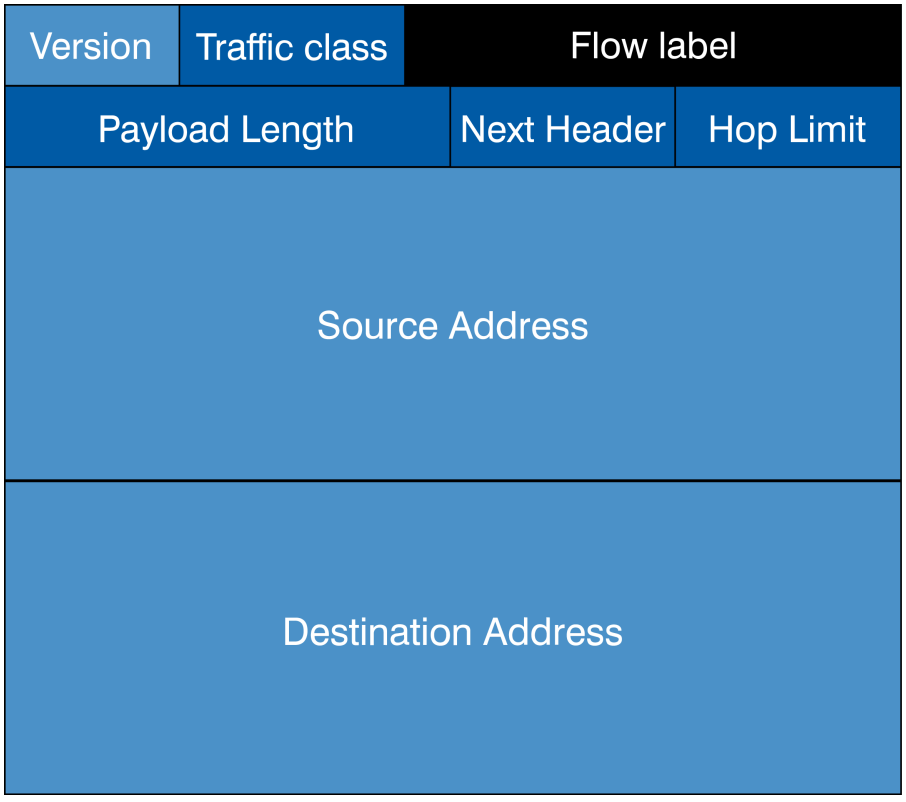
23.3.1 Internet Protocol header comparison

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



IPv6 Header



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.

23.3.2 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.

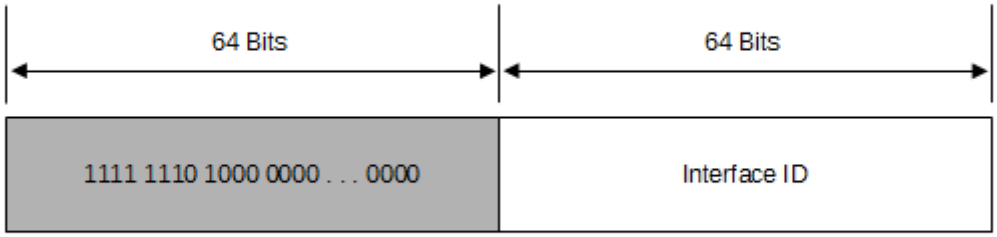
IPv6 addresses may be abbreviated to shorter notations. The following rules for abbreviation are defined by RFC 5952 *"A Recommendation for IPv6 Address Text Representation"*.

- One or more leading zeros from any groups of hexadecimal digits are removed; this is usually done to either all or none of the leading zeros. For example, the group 0222 is converted to 222.
- Consecutive sections of zeros 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`.

23.3.2.1 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.

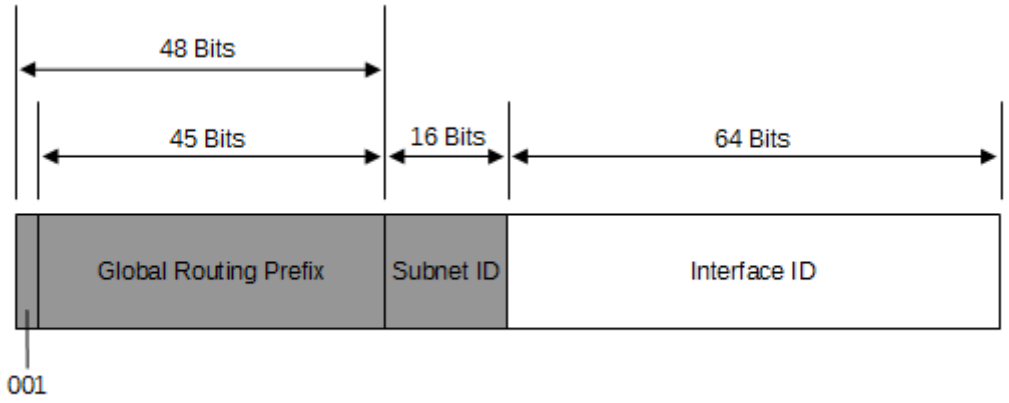


The first 64 bits are link-local address prefix. Prefixes for IPv6 subnets are expressed in the same way as Classless Inter-Domain Routing (CIDR) notation for IPv4. An IPv6 prefix is written in address/prefix-length notation. The prefix for link-local addresses is always `fe80::/64`.

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.

23.3.2.2 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 Inter- net. According to RFC 3587 an IPv6 global unicast address consists of 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 net- work 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.

23.3.3 Further reading for IPv6

This chapter explains the usage of the emNet 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 emNet 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.

23.3.3.1 IPv6 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#	Description
[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*

23.3.3.2 Related books for IPv6

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

23.4 Include IPv6 to your emNet start project

Integration of emNet is a relatively simple process, which consists of the follow- ing steps:

- Step 1: Open an emNet project and compile it.
- Step 2: Add emNet 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.

23.4.1 Open an emNet project and compile it

To add the emNet IPv6 add-on to your project, you need a running emNet project. For a step by step tutorial to setup an emNet project refer to *Running emNet on target hardware* on page 56.

23.4.2 Add the emNet IPv6 add-on to the start project

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

- IP

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

23.4.2.1 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()` in the function `IP_X_Config()` as shown below:

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

...
  IFaceId = IP_AddEtherInterface(DRIVER);   // Add driver for your hardware.
...
  //
  // Add protocols to the stack.
  //
  IP_TCP_Add();
  IP_UDP_Add();
  IP_ICMP_Add();
#if IP_SUPPORT_IPV6
  IP_IPV6_Add(IFaceId);
#endif
```

Refer to *Configuring emNet* on page 614 for more information on `IP_X_Config()`.

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.

23.4.2.2 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.

emNet sample configurations up to emNet 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
```

23.4.2.3 Enable terminal output for IPv6 messages

In debug builds of emNet, 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
                );
```

23.4.2.4 Select the start application

For testing of your emNet 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`)

23.4.3 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.

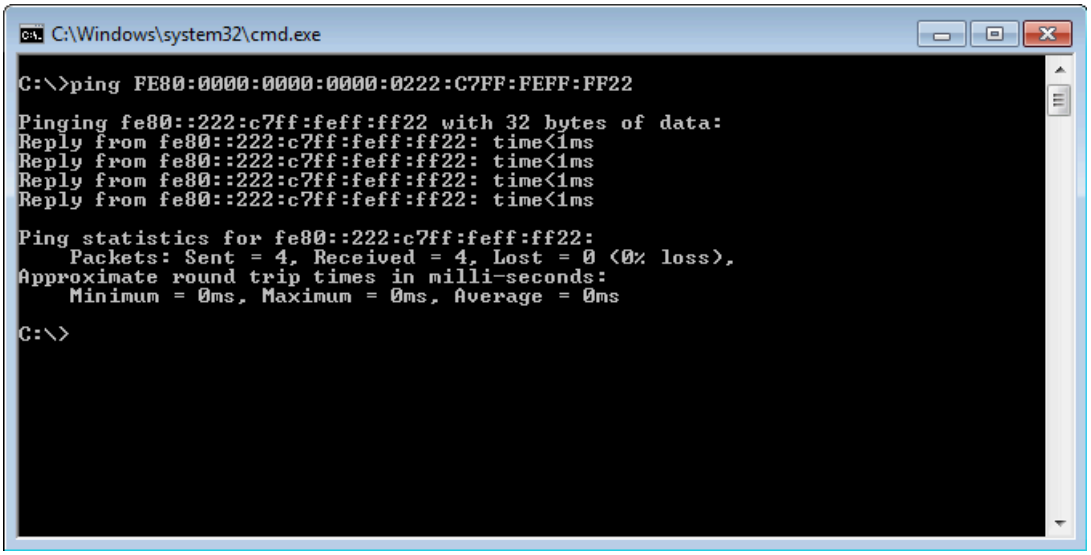
A target which uses IPv4 and IPv6 should output similar logging messages as shown below:

```
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.:
 FE80:0000:0000:0000:0222:C7FF:FEFF:FF22 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.



23.5 Configuration

The emNet 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.

23.5.1 IPv6 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).

23.5.2 IPv6 Compile time configuration switches

Type	Symbolic name	Default	Description
B	IP_SUPPORT_IPV6	0	Enables support for IPv6. Refer to *IP_IPV6_Add* on page 640 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.

23.5.3 IPv6 Runtime configuration

Please refer to *IP_IPV6_Add* on page 640 for detailed information about runtime configuration.

23.6 IPv6 API functions

Function	Description
Configuration functions	
IP_IPV6_Add()	Adds IPv6 to the stack and initializes the specified interface.
IP_IPV6_AddUnicastAddr()	Adds an additional IPv6 unicast (or anycast) address to the interface.
IP_IPV6_ChangeDefaultConfig()	Changes the default IPv6 configuration of the stack.
IP_IPV6_GetIPv6Addr()	Returns an IPv6 address and optionally the number of configured IPv6 address of the selected interface.
IP_IPV6_GetIPPacketInfo()	Returns the start address of the data part of an IPv6 packet.
IP_IPV6_ParseIPv6Addr()	Transforms an IPv6 address separated by colons into a byte stream (big endian byte order).
IP_IPV6_SetDefHopLimit()	Adds a default hop limit to the interface.
IP_IPV6_SetGateway()	Sets the gateway server address of the selected interface.
IP_IPV6_SetLinkLocalUnicastAddr()	Adds a link local unicast address to the interface.
IP_IPV6_INFO_GetConnectionInfo()	Retrieves the connection information for a connection handle.
IP_ICMPV6_AddRxHook()	This function adds a callback that is executed upon receiving an ICMPv6 packet.
IP_ICMPV6_MLD_AddMulticastAddr()	Adds an additional multicast address to the given interface.
IP_ICMPV6_MLD_RemoveMulticastAddr()	Removes a multicast address from the given interface.
IP_ICMPV6_NDP_SetDNSSLCallback()	Sets a callback to allow processing of DNS Search List Option.
IP_IPV6_ResolveHost()	Resolves an IP addr.

23.6.1 IP_IPV6_Add()

Description

Adds IPv6 to the stack and initializes the specified interface.

Prototype

```
void IP_IPV6_Add(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

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\x22");        // 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(0);  // Add IPv6, includes ICMPv6, MLD and NDP on interface 0.
```



```
}
```


23.6.2 IP_IPV6_AddUnicastAddr()

Description

Adds an additional IPv6 unicast (or anycast) address to the interface.

Prototype

```
int IP_IPV6_AddUnicastAddr(      unsigned   IFaceId,
                           const U8       * pAddr);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pAddr	Pointer to the 16 byte IPv6 address which should be added to the network interface.

Return value

0 OK. IPv6 address added to the network interface.
1 Error. IPv6 address could not be added to the network interface.

Additional information

Normally, a link-local address (prefix FE80::/64) derived from the MAC address of the interface has been set during initialization of the stack.

23.6.3 IP_IPV6_ChangeDefaultConfig()

Description

Changes the default IPv6 configuration of the stack.

Prototype

```
void IP_IPV6_ChangeDefaultConfig(unsigned IFaceId,
                                 unsigned Option,
                                 unsigned OnOff);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
Option	Option to enable/disable.
OnOff	• 1: Enable option • 0: Disable option

Additional information

In most situations, the default configuration of the stack needs no changes. To adapt the behavior of the stack for the needs of some special use cases, the default behavior can be changed.

Options

Option	Description
IPV6_GENERATE_LINK_LOCAL_ADDR	Generate a link-local address fe80::/64
IPV6_ICMPV6_MLD_ADD_DEF_ADDR	Add multicast addresses ALL-Nodes and All-Router to the interface.
IPV6_USE_SLAAC	Use Stateless Address Autoconfiguration (SLAAC).
IPV6_USE_ROUTER_ADVERTISMENTS	Use configuration options supplied through Router Advertisements.

23.6.4 IP_IPV6_GetIPv6Addr()

Description

Returns an IPv6 address and optionally the number of configured IPv6 address of the selected interface.

Prototype

```
void IP_IPV6_GetIPv6Addr(unsigned    IFaceId,
                         U8          AddrId,
                         IPV6_ADDR * pIPv6Addr,
                         U8        * pNumAddr);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface identifier.
AddrId	Address index.
pIPv6Addr	IPv6 address for the result (Could be NULL).
pNumAddr	Number of configured IPv6 addresses (Could be NULL).

23.6.5 IP_IPV6_GetIPPacketInfo()

Description

Returns the start address of the data part of an IPv6 packet.

Prototype

```
U8 *IP_IPV6_GetIPPacketInfo(IP_PACKET * pPacket);
```

Parameters

Parameter	Description
pPacket	Pointer to an IP_PACKET .

Return value

≠ NULL Start address of the data part of the IPv6 packet.
= NULL Error.

Example

```
/*********************************************************************
*
*       _pfOnRxICMP
*/
static int _pfOnRxICMP(IP_PACKET* pPacket) {
  const U8* pData;

  pData = IP_IPV6_GetIPPacketInfo(pPacket);
  if(*pData == 128u) {
    printf("ICMPv6 echo request received!\n");
  }
  if(*pData == 129u) {
    printf("ICMPv6 echo reply received!\n");
  }
  return 0;
}
```


23.6.6 IP_IPV6_ParseIPv6Addr()

Description

Transforms an IPv6 address separated by colons into a byte stream (big endian byte order).

Prototype

```
int IP_IPV6_ParseIPv6Addr(const char      * sHost,
                          IPV6_ADDR * pIPv6Addr);
```

Parameters

Parameter	Description
sHost	Pointer to the IPv6 address string to parse.
pIPv6Addr	Pointer to an IPv6 address structure to store the converted byte stream.

Return value

0	OK.
-1	Error. Not every character in address are hexa values (0-f) or colons (:).
-2	Error. Too many characters for 16bit block.
-3	Error. Illegal number of colons in a row (“:::”)
-4	Error. “::” is used twice.
-5	Error. Address string to long.
-6	Error. Too many 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.

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
```


23.6.7 IP_IPV6_SetDefHopLimit()

Description

Adds a default hop limit to the interface.

Prototype

```
int IP_IPV6_SetDefHopLimit(unsigned IFaceId,
                           U8       HopLimit);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
HopLimit	Hop limit that should be used as default.

Return value

0	Ok. Hop limit set.
1	Error.

23.6.8 IP_IPV6_SetGateway()

Description

Sets the gateway server address of the selected interface.

Prototype

```
int IP_IPV6_SetGateway(      unsigned   IFaceId,
                       const char     * sIFaceAddr,
                       const char     * sRouterAddr);
```

Parameters

Parameter	Description
IFaceId	Interface identifier
sIFaceAddr	IPv6 address string (interface IP). For example “2001:4860:4860::4444”
sRouterAddr	IPv6 address string (router IP). For example “2001:4860:4860::8888”

Return value

< 0	Error. Gateway address format invalid.
= 0	Could not add gateway. - IPv6 not enabled on the selected interface or gateway is already in list or required memory could not allocated.
= 1	Gateway added.

23.6.9 IP_IPV6_SetLinkLocalUnicastAddr()

Description

Adds a link local unicast address to the interface.

Prototype

```
void IP_IPV6_SetLinkLocalUnicastAddr(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface identifier.

23.6.10 IP_IPV6_INFO_GetConnectionInfo()

Description

Retrieves the connection information for a connection handle. The connetion handle is typically obtained via a call to `IP_INFO_GetConnectionList()`.

Prototype

```
int IP_IPV6_INFO_GetConnectionInfo(IP_CONNECTION_HANDLE   hConn,
                                   IP_IPV6_CONNECTION   * pConInfo);
```

Parameters

Parameter	Description
`hConn`	Connection handle.
`pConInfo`	Pointer on an `IP_CONNECTION` structure that will be filled.

Return value

0 OK, Information retrieved
1 Error, typically because the connection pointer is no longer in list

23.6.11 IP_ICMPV6_AddRxHook()

Description

This function adds a callback that is executed upon receiving an ICMPv6 packet.

Prototype

```
void IP_ICMPV6_AddRxHook(IP_HOOK_ON_ICMPV6 * pHook,
                         IP_ON_ICMPV6_FUNC * pf,
                         void              * pUserContext);
```

Parameters

Parameter	Description
pHook	Pointer to static element of IP_HOOK_ON_ICMPV4 that can be internally used by the stack.
pf	Function pointer to the callback to execute.
pUserContext	User defined context top pass to the callback.

Example

```
static IP_HOOK_ON_ICMPV6 _Hook;

/*********************************************************************
*
*       _cbOnRx()
*
*  Function description
*    Callback executed when an ICMPv6 packet is received.
*
*  Parameters
*    IFaceId     : Zero-based interface index.
*    pPacket     : Packet that has been received.
*    pUserContext: User context given when adding the hook.
*    p           : Reserved for future extensions of this API.
*
*  Return value
*    == IP_OK                   : Packet has been handled (freed or reused).
*    == IP_OK_TRY_OTHER_HANDLER: Packet is untouched and stack shall try another
 handler.
*
*  Additional information
*    The callback can remove its own hook using IP_ICMPV6_RemoveRxHook() .
*/
static int _cbOnRx(unsigned   IFaceId,
                   IP_PACKET* pPacket,
                   void*      pUserContext,
                   void*      p) {
  const U8* pData;

  IP_USE_PARA(IFaceId);
  IP_USE_PARA(pUserContext);
  IP_USE_PARA(p);

  pData = IP_IPV6_GetIPPacketInfo(pPacket);
  if(*pData == IP_ICMPV6_TYPE_ECHO_REQUEST) {
    IP_Logf_Application("ICMPv6 echo request received!");
  }
  if(*pData == IP_ICMPV6_TYPE_ECHO_REPLY) {
    IP_Logf_Application("ICMPv6 echo reply received!");
  }
  //
  // Optional: Remove the hook once no longer needed.
```



```
  //
  IP_ICMPV6_RemoveRxHook(SEGGER_PTR2PTR(IP_HOOK_ON_ICMPV6, pUserContext));
  return IP_OK_TRY_OTHER_HANDLER;  // Let the stack handle the message.
}

/*********************************************************************
*
*  MainTask()
*
*  Function description
*    Main task executed by the RTOS to create further resources and
*    running the main application.
*/
void MainTask(void) {
 IP_Init();
 //
 // Add a hook that gets notified about received ICMP messages.
 // In this example the pointer to the hook item itself is passed as
 // user context to demonstrate the hook removing itself.
 //
 IP_ICMPV6_AddRxHook(&_Hook, _cbOnRx, &_Hook);
 ...
}
```


23.6.12 IP_ICMPV6_RemoveRxHook()

Description

This function removes a hook function from the IP_HOOK_ON_ICMPV6 list.

Prototype

```
void IP_ICMPV6_RemoveRxHook(IP_HOOK_ON_ICMPV6 * pHook);
```

Parameters

Parameter	Description
pHook	Element of type IP_HOOK_ON_ICMPV6 to remove from list.

23.6.13 IP_ICMPV6_MLD_AddMulticastAddr()

Description

Adds an additional multicast address to the given interface.

Prototype

```
int IP_ICMPV6_MLD_AddMulticastAddr(unsigned    IFaceId,
                                   IPV6_ADDR * pMultiCAddr);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pMultiCAddr	Pointer to the 16 byte IPv6 multicast address which should be added to the network interface.

Return value

1 OK. IPv6 multicast address added to the network interface.
0 Error. IPv6 multicast address could not be added to the network interface.

Additional information

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 functioning of the IPv6 implementation.

23.6.14 IP_ICMPV6_MLD_RemoveMulticastAddr()

Description

Removes a multicast address from the given interface.

Prototype

```
int IP_ICMPV6_MLD_RemoveMulticastAddr(unsigned    IFaceId,
                                      IPV6_ADDR * pIPv6Addr);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pIPv6Addr	Pointer to the 16 byte IPv6 multicast address which should be removed.

Return value

1 OK. IPv6 multicast address removed from the network interface.
0 Error. IPv6 multicast address could not be removed.

23.6.15 IP_ICMPV6_NDP_SetDNSSLCallback()

Description

Sets a callback to allow processing of DNS Search List Option. (For further information to the DNS Search List Option refert to [6], section 5.2.

Prototype

```
void IP_ICMPV6_NDP_SetDNSSLCallback
      (unsigned IFaceId,
       void     ( *pfHandleDNSSLOpt)
(unsigned IFaceId , U8 * pData , unsigned NumBytes , U32 Lifetime ));
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pfHandleDNSSLOpt	Callback with more information.

23.6.16 IP_IPV6_ResolveHost()

Description

Resolves an IP addr. given as text string into an actual IP addr.

Prototype

```
int IP_IPV6_ResolveHost(      unsigned    IFaceId,
                        const char      * sHost,
                              IPV6_ADDR * pIPv6Addr,
                              U32         ms);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
sHost	Host name string to resolve.
pIPv6Addr	Pointer where to store the resolved IP addr. in network byte order.
ms	Timeout for DNS request [ms] if the request can not be sent immediately.

Return value

O.K. : = 0 Error: < 0

23.7 IPv6 internal functions, variables and data-structures

emNet internal functions, variables and data-structures are not explained here as they are in no way required to use emNet. 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 emNet. The following data-structures are meant for public usage together with the documented API.

23.7.1 IP_ON_ICMPV6_FUNC

Description

Callback executed when an ICMPv6 packet is received.

Type definition

```
typedef int (IP_ON_ICMPV6_FUNC)(unsigned    IFaceId,
                                 IP_PACKET * pPacket,
                                 void      * pUserContext,
                                 void      * p);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pPacket	Packet that has been received.
pUserContext	User context given when adding the hook.
p	Reserved for future extensions of this API.

Return value

IP_OK	Packet has been handled (freed or reused).
IP_OK_TRY_OTHER_HANDLER	Packet is untouched and stack shall try another handler.

Additional information

The callback can remove its own hook using IP_ICMPV6_RemoveRxHook() .

23.8 IPv6 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 emNet IPv6 add-on are the following extensions included.

Structures

Structures	Description
sockaddr_in6	Structure to handle IPv6 addresses.

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.

23.8.1 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.

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 661 and *IPv6 API functions* on page 639 for details about the usage of the socket API functions.

23.9 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 emNet IPv6 add-on shipment.

23.9.1 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 660. emNet IPv6 add-on comes with three version of `OS_IP_SimpleServer`.

File	Description
`OS_IP_SimpleServer.c`	IPv4 version of the simple TCP server example. Server listens on port 23 for IPv4 clients.
`OS_IP_SimpleServer_IPv6.c`	IPv6 version of the simple TCP server example. Server listens on port 23 for IPv6 clients.
`OS_IP_SimpleServer_IPv4_IPv6.c`	Dual stack version of the simple TCP server example. Server listens on port 23 for IPv4 and IPv6 clients.

23.9.1.1 TCP/IPv4 server sample code

Listen to a port

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 *sockaddr_in* on page 348.

Accept connection

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 emNet 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 299.

23.9.1.2 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.

Listen to a port

_ListenAtTcpAddr() needs to create an IPv6 socket instead of an IPv4 socket and the sockaddr_in structure has to replaced by a sockaddr_in6 structure.

The socket creation changes from:

```
hSock = socket(AF_INET, 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.

Accept connection

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.

23.9.1.3 Dual stack TCP server sample code

The emNet IPv6 add-on provides IPv4 and IPv6 protocol stacks in the same net- work node. This means that emNet 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 317.

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.

23.10 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.

23.10.1 IPv6 ROM usage

The resource usage of the IPv6 add-on has been measured on a Cortex-M3 system size optimization.

Addon	ROM
emNet IPv6	approximately 8.0 kBytes

The stated ROM usage is only the additional space that is required to add IPv6 to the emNet IPv4 stack. The total ROM usage for emNet running IPv4 and IPv6 is approximately 28 kBytes.

23.10.2 RAM usage

The total memory requirements of the IPv6 add-on can basically be computed as the sum of the following components:

Description	RAM
IPv6 add-on	approximately 200 bytes
Unicast address	n * approximately 48 bytes
Multicast address	n * approximately 28 bytes
NDP entry	n * 52 bytes

An IPv6 target with two unicast addresses, four Multicast address and five NDP entries needs approximately 660 bytes additional RAM. For detailed information about the configuration and the memory requirements for each TCP/UDP connection, refer to *Configuring emNet* on page 614.

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 83.

Chapter 24

SMTP client (Add-on)

The emNet SMTP client is an optional extension to emNet. The SMTP client can be used with emNet 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.

24.1 emNet SMTP client

The emNet SMTP client is an optional extension which can be seamlessly integrated into your TCP/IP application. It combines a maximum of performance with a small memory footprint. The SMTP client allows an embedded system to send emails with dynamically generated content. The RAM usage of the SMTP client module has been kept to a minimum by smart buffer handling.

The SMTP client implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 821]	Simple Mail Transfer Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc821.txt*
[RFC 974]	Mail routing and the domain system Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc974.txt*
[RFC 2554]	SMTP Service Extension for Authentication Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc2554.txt*
[RFC 5321]	Simple Mail Transfer Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc5321.txt*

The following table shows the contents of the emNet SMTP client root directory:

Directory	Content
`.\Application\`	Contains the example application to run the SMTP client with emNet.
`.\Config\`	Contains the SMTP client configuration file. Refer to *SMTP client configuration* on page 678 for detailed information.
`.\Inc\`	Contains the required include files.
`.\IP\`	Contains the SMTP client sources, `IP_SMTPC.c` and `IP_SMTPC.h`.
`.\Windows\SMTPC\`	Contains the source, the project files and an executable to run emNet SMTP client on a Microsoft Windows host.

24.2 Feature list

- Low memory footprint.
- Independent of the TCP/IP stack: any stack with sockets can be used.
- Support for attachments (multipart items).
- Supports AUTH TLS (with additional SSL/TLS stack) for secure connections.
- Independent of the SSL/TLS stack: any sockets based stack can be used.
- Example applications included.
- Project for executable on PC for Microsoft Visual Studio included.

24.3 Requirements

TCP/IP stack

The emNet SMTP client requires a TCP/IP stack. It is optimized for emNet, 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 emNet.

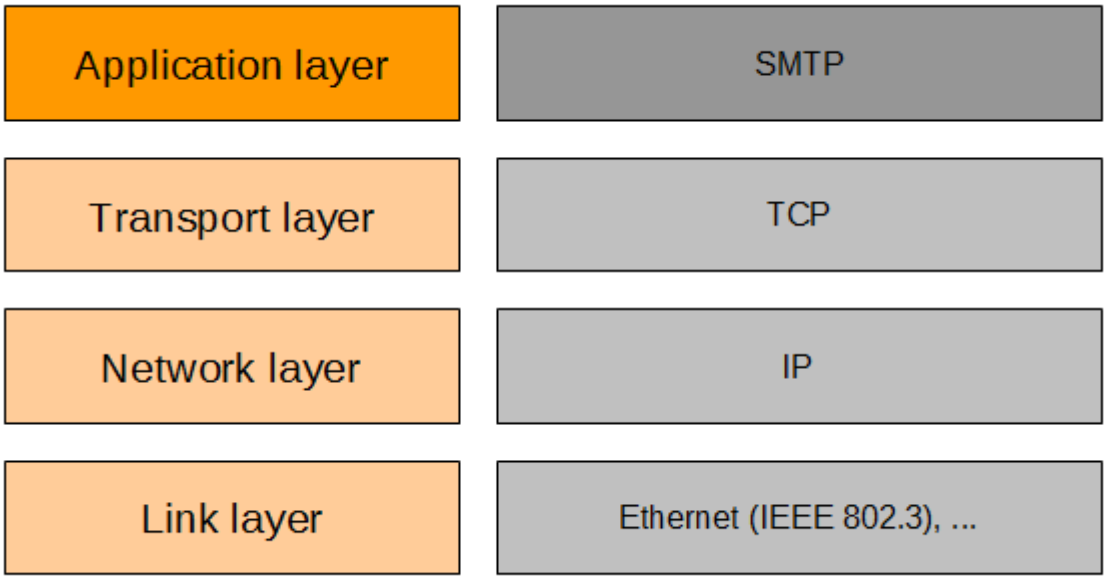
In addition to insecure connections the emNet SMTP client API allows to secure the connection by using an additional SSL/TLS stack if the server is capable of the STARTTLS command.

Multi tasking

The SMTP client needs to run as a separate thread. Therefore, a multi tasking system is required to use the emNet SMTP client.

24.4 SMTP backgrounds

The Simple Mail Transfer Protocol is a text based communication protocol for electronic mail transmission across IP networks.



Using SMTP, an emNet 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 emNet 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 emNet SMTP client supports only PLAIN and 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
```


24.5 Secure connections

Most modern mail servers support secure connections or might even refuse insecure connections at all. There are two ways of establishing secure connections with a mail server by utilizing an additional SSL/TLS stack:

- Establishing an SMTPS SSL/TLS connection on port 465 where the whole connection from the first data byte sent to the last byte is secured. This method is deprecated but still widely in use.
- Establish a regular insecure connection on port 25 and try to upgrade it to a secure connection using the STARTTLS command.

While the second method (starting insecure and trying to upgrade) might look less secure it is not. The security of this method is defined by the configured fallback mechanism used via the `SecPolicy` that can be found in *Structure `IP_SMTPC_MTA`* on page 690.

If the client is not allowed to use an insecure connection as fallback this method is as safe as encrypting the whole connection from start to end. An advantage of this method is that most firewalls will allow traffic on port 25 and existing firewall rules do not have to changed for secure/insecure connections.

For an example on how to use secure connections please refer to the `IP_SendMail_Secure.c` sample.

24.6 Attachments

Mail attachments are multipart items that are added to a mail. Multipart items in a mail are not limited to attachments only but this is the most common form they are used.

The emNet SMTPC client allows to easily add attachments to a mail by providing an array of `IP_SMTPC_MULTIPART_ITEM` items that configure the attachments to add. For more information about the item structure please refer to *Structure `IP_SMTPC_MULTIPART_API`* on page 687.

Creating a list of attachments

Each item is sent via a callback that needs to be implemented by the application. Multiple items can share the same callback. To differentiate between two items that use the same callback a user context element can be set in the `IP_SMTPC_MULTIPART_ITEM`. This context can be a simple index or a pointer to more information about the item to send like a string with a path of a file to send from a file system.

The content type set for the item defines how the receiver will treat this attachment. The following is only a short excerpt of possible content types (MIME types) that can be used but. However all depend on being known by the receiver to be fully under- stood:

File extension	Content type (MIME type)
.jpg	“image/jpeg”
.txt	“text/plain”
.zip	“application/x-compressed”

One extension might be known under multiple MIME types. The sender usually chooses the MIME type that is known to the system for this extension. Therefore an attachment might not be correctly recognized by the receiver as this type of file under all circumstances.

Adding the attachments to the mail

To actually add the items to the mail the list has to be assigned to the message to send. This is done by filling in the `sBoundary`, `paMultipartItem` and `NumMultipartItems` parameters of the `IP_SMTPC_MESSAGE` structure that defines the message to send. For more information about the structure defining a message please refer to *Structure `IP_SMTPC_MESSAGE`* on page 689.

A sample of adding two text files as attachment can be found in the `IP_SendMail.c` sample.

24.7 SMTP client configuration

The emNet 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).

24.7.1 SMTP client compile time configuration switches

Type	Symbolic name	Default	Description
F	SMTPC_WARN	--	Defines a function to output warnings. In debug configurations (DEBUG = 1) SMTPC_WARN maps to IP_Warnf_Application().
F	SMTPC_LOG	--	Defines a function to output logging messages. In debug configurations (DEBUG = 1) SMTPC_LOG maps to IP_Logf_Application().
N	SMTPC_SERVER_PORT	25	Defines the port where the SMTP server is listening.
N	SMTPC_IN_BUFFER_SIZE	256	Defines the size of the input buffer. The input buffer is used to store the SMTP replies of the SMTP server.
N	SMTPC_AUTH_USER_BUFFER_SIZE	48	Defines the size of the buffer used for the Base-64 encoded user name.
N	SMTPC_AUTH_PASS_BUFFER_SIZE	48	Defines the size of the buffer used for the Base-64 encoded password.

24.8 API functions

Function	Description
IP_SMTPC_Send()	Sends an email to one or multiple recipients.

24.8.1 IP_SMTPC_Send()

Description

Sends an email to one or multiple recipients.

Prototype

```
int IP_SMTPC_Send(const IP_SMTPC_API         * pIP_API,
                  const IP_SMTPC_MAIL_ADDR   * paMailAddr,
                        int                    NumMailAddr,
                  const IP_SMTPC_MESSAGE     * pMessage,
                  const IP_SMTPC_MTA         * pMTA,
                  const IP_SMTPC_APPLICATION * pApplication);
```

Parameters

Parameter	Description
pIP_API	Pointer to an IP_STMPC_API structure.
paMailAddr	Pointer to an array of IP_SMTPC_MAIL_ADDR structures. 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	Number of email addresses.
pMessage	Pointer to an array of IP_SMTPC_MESSAGE structures.
pMTA	Pointer to an array of IP_SMTPC_MTA structures.
pApplication	Pointer to an array of IP_SMTPC_APPLICATION structures.

Return value

0	OK.
1	Error.

24.9 Data structures

Structure	Description
IP_SMTPC_API	Structure with pointers to the required socket interface functions.
IP_SMTPC_APPLICATION	Structure with application related elements.
IP_SMTPC_MULTIPART_API	Structure with API for sending attachments.
IP_SMTPC_MULTIPART_ITEM	Structure defining one item to attach.
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.

24.9.1 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);
  int          (*pfUpgrade)   (SMTPC_SOCKET hSock,
                               const char * sServer);
  void         (*pfDowngrade) (SMTPC_SOCKET hSock);
} IP_SMTPC_API;
```

Member	Description
pfConnect	Pointer to the connect function (for example, `connect()`).
pfDisconnect	Pointer to the disconnect function (for example, `closesocket()`).
pfSend	Pointer to a send function (for example, `send()`).
pfReceive	Pointer to a receive function (for example, `recv()`).
pfUpgrade	Pointer to a callback function to upgrade an insecure connection to a secure connection using an SSL/TLS stack.
pfDowngrade	Pointer to a callback function to downgrade a secure connection to an insecure connection (after work is done) using an SSL/TLS stack.

Example

```
/*********************************************************************
*
*       _Connect
*
*  Function description
*    Creates a socket and opens a TCP connection to the mail host.
*/
static SMTPC_SOCKET _Connect(char * SrvAddr) {
  long IP;
  long Sock;
  struct hostent * pHostEntry;
  struct sockaddr_in sin;
  int r;

  //
  // Convert host into mail host
  //
  pHostEntry = gethostbyname(SrvAddr);
  if (pHostEntry == NULL) {
    SMTPC_LOG(("gethostbyname failed: %s\r\n", SrvAddr));
    return NULL;
  }
  IP = *(unsigned*)(*pHostEntry->h_addr_list);
  //
  // Create socket and connect to mail server
  //
  Sock = socket(AF_INET, SOCK_STREAM, 0);
  if(Sock  == -1) {
    SMTPC_LOG(("Could not create socket!"));
    return NULL;
```



```
  }
  IP_MEMSET(&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_port = htons(SERVER_PORT);
  sin.sin_addr.s_addr = IP;
  r = connect(Sock, (struct sockaddr*)&sin, sizeof(sin));
  if(r == SOCKET_ERROR) {
    SMTPC_LOG(("\nSocket error :"));
    return NULL;
  }
  SMTPC_LOG(("APP: Connected.\r\n"));
  return (SMTPC_SOCKET)Sock;
}

/*********************************************************************
*
*       _Disconnect
*
*  Function description
*    Closes a socket.
*/
static void _Disconnect(SMTPC_SOCKET Socket) {
  closesocket((long)Socket);
}

/*********************************************************************
*
*       _Send
*
*  Function description
*    Sends data via socket interface.
*/
static int _Send(const char * buf, int len, void * pConnectionInfo) {
  return send((long)pConnectionInfo, buf, len, 0);
}

/*********************************************************************
*
*       _Recv
*
*  Function description
*    Receives data via socket interface.
*/
static int _Recv(char * buf, int len, void * pConnectionInfo) {
  return recv((long)pConnectionInfo, buf, len, 0);
}

static const IP_SMTPC_API _IP_Api = {
  _Connect,
  _Disconnect,
  _Send,
  _Recv
};
```


24.9.2 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, sample.com. 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: "+0100". Can be NULL.

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.
  NULL
};
```


24.9.3 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.

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 680.

The sName could also be a UTF-8 encoded string. For example, to send “Лев Толстой” define sName as

```
"\xD0\x9B\xD0\xB5\xD0\xB2\x20\xD0\xA2\xD0\xBE\xD0\xBB\xD1\x81\xD1\x82\xD0\xBE
\xD0\xB9"
```

For more details on UTF-8 encoding refer for example to https://en.wikipedia.org/wiki/Utf8.

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 emNet SMTP client";
  Message.sBody    = "emNet 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 recipient(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);
}
```


24.9.4 Structure IP_SMTPC_MULTIPART_API

Description

Structure containing functions for sending multipart content (attachments).

Prototype

```
typedef struct {
  void (*pfSend)(       IP_SMTPC_CONTEXT* pContext,
                  const char*             pData,
                        unsigned          NumBytes);
} IP_SMTPC_MULTIPART_API;
```

Member	Description
pfSend	Callback for sending multipart content in chunks.
- pContext	SMTPc context.
- pData	Data to send.
- NumBytes	Amount of data to send.

24.9.5 Structure IP_SMTPC_MULTIPART_ITEM

Description

Structure to store multipart items to be added to the mail.

Prototype

```
typedef struct IP_SMTPC_MULTIPART_ITEM_STRUCT IP_SMTPC_MULTIPART_ITEM;
struct IP_SMTPC_MULTIPART_ITEM_STRUCT {
  const char* sFilename;
  const void* pUserContext;
  const char* sContentType;
  void (*pfSendItem)(       IP_SMTPC_CONTEXT*        pContext,
                      const IP_SMTPC_MULTIPART_ITEM* pItem,
                      const IP_SMTPC_MULTIPART_API*  pAPI ));
};
```

Member	Description
sFilename	Filename suggested to client in case the multipart item is an attachment.
pUserContext	User context passed to pfSendItem callback. Can be used to pass a file path or other application information to the callback.
sContentType	Value for the “Content-Type: ” field of the multipart item. Examples for a text file attachment are: “text/plain” or “text/plain; name=Test.txt” .
pfSendItem	Callback for sending the content of a multipart item without having to know its length upfront.
- pContext	SMTPc context.
- pItem	Item to send.
pAPI	API for sending data in chunks.

24.9.6 Structure IP_SMTPC_MESSAGE

Description

Structure to store the subject and the text and attachments of the email.

Prototype

```
typedef struct {
  const char*                    sSubject;
  const char*                    sBody;
  const char*                    sBoundary;
  const IP_SMTPC_MULTIPART_ITEM* paMultipartItem;
        unsigned                 NumMultipartItems;
} IP_SMTPC_MESSAGE;
```

Member	Description
sSubject	Subject of the message to send. Could be UTF-8 encoded string.
sBody	Content of the message to send. Could be UTF-8 encoded string.
sBoundary	Boundary to use for multipart encoding (e.g. when using attachments). Can be NULL if (NumMultipartItems = 0).
paMultipartItem	Pointer to list of items (attachments) to multipart encode with the message. Can be NULL if (NumMultipartItems = 0).
NumMultipartItems	Number of multipart items that can be found at paMultipartItem.

24.9.7 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;
        U8     SecPolicy;
} IP_SMTPC_MTA;
```

Member	Description
sServer	Server address (for example, "mail.foobar.com").
sUser	Account user name (for example, "foo@bar.com"). Can be NULL.
sPass	Account password (for example, "password"). Can be NULL.
SecPolicy	Security policy to use: - SMTPC_SEC_POLICY_ALLOW_INSECURE (default) - SMTPC_SEC_POLICY_SECURE_ONLY

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 or AUTH PLAIN command to the server. If the server does not support authentication, he will return an error code and the client closes the session.

The parameter SecPolicy defines the fallback behavior in case secure connections are supported by using an additional SSL/TLS stack and providing callbacks for the function pointers for pfUpgrade and pfDowngrade in the *Structure IP_SMTPC_API* on page 682.

SMTPC_SEC_POLICY_ALLOW_INSECURE (default) will allow a fallback to an insecure connection to be used in case no SSL/TLS stack is available or the server does not offer the AUTH TLS extension.

If a secure connection is possible as the server offers the AUTH TLS connection a secure connection will be established.

SMTPC_SEC_POLICY_SECURE_ONLY forces the SMTP client to insist on a secure connection and will refuse to proceed using an insecure connection.

24.10 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 a Cortex-M4 system. Details about the further configuration can be found in the sections of the specific example.

Configuration used

```
#define SMTPC_OUT_BUFFER_SIZE        256
#define SMTPC_IN_BUFFER_SIZE         256
#define SMTPC_AUTH_USER_BUFFER_SIZE   48
#define SMTPC_AUTH_PASS_BUFFER_SIZE   48
```

24.10.1 ROM usage on a Cortex-M4 system

The following resource usage has been measured on a Cortex-M4 system using SEGGER Embedded Studio V3.10e, size optimization.

Addon	ROM
emNet SMTP client	approximately 2.7 kBytes

24.10.2 RAM usage

Addon	RAM
emNet SMTP client buffers w/o task stack	approximately 600 Bytes

The RAM requirement for the work buffers on the task stack is approximately 600 bytes for the mentioned configuration. Only the bigger buffer size of `SMTPC_AUTH_USER_BUFFER_SIZE` or `SMTPC_AUTH_PASS_BUFFER_SIZE` is used for a single buffer.

Chapter 25

emFTP server (Add-on)

The emFTP server is an optional extension to the emNet TCP/IP stack. The emFTP server can be used with emNet 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.

25.1 emFTP server

The emFTP 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 emFTP 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 emFTP server root directory:

Directory	Content
`.\Application\`	Contains the example application to run the FTP server with emNet.
`.\Config\`	Contains the FTP server configuration file.
`.\Inc\`	Contains the required include files.
`.\IP\`	Contains the FTP server sources.
`.\IP\FS\`	Contains the sources for the file system abstraction layer and the read-only file system. Refer to *File system abstraction layer* on page 1251 for detailed information.
`.\Windows\FTPserver\`	Contains the source, the project files and an executable to run emFTP server on a Microsoft Windows host.

25.2 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.

25.3 Requirements

TCP/IP stack

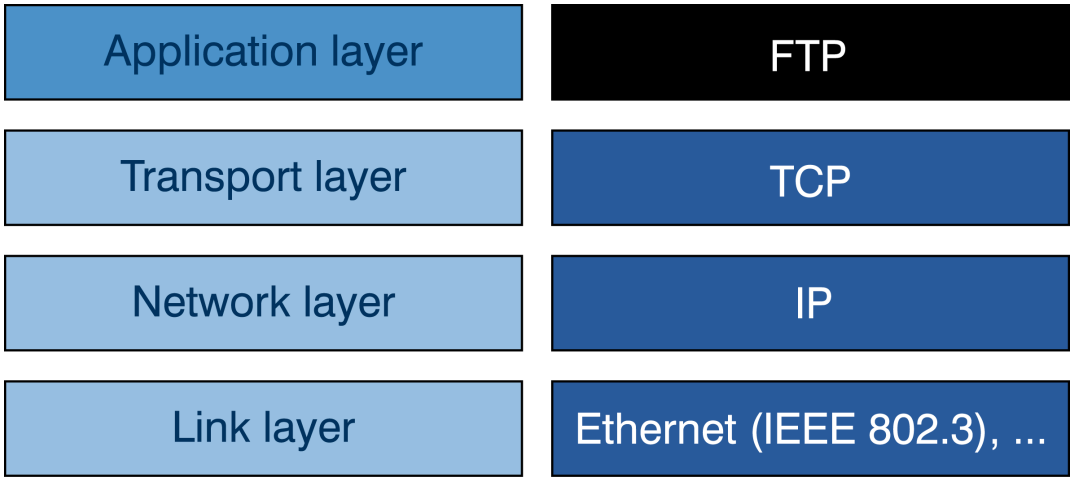
The emFTP server requires a TCP/IP stack. It is optimized for emNet, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and two implementations which use the socket API of emNet (with or without TLS secured connection).

Multi tasking

The FTP server needs to run as a separate thread. Therefore, a multi tasking system is required to use the emFTP server.

25.4 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.

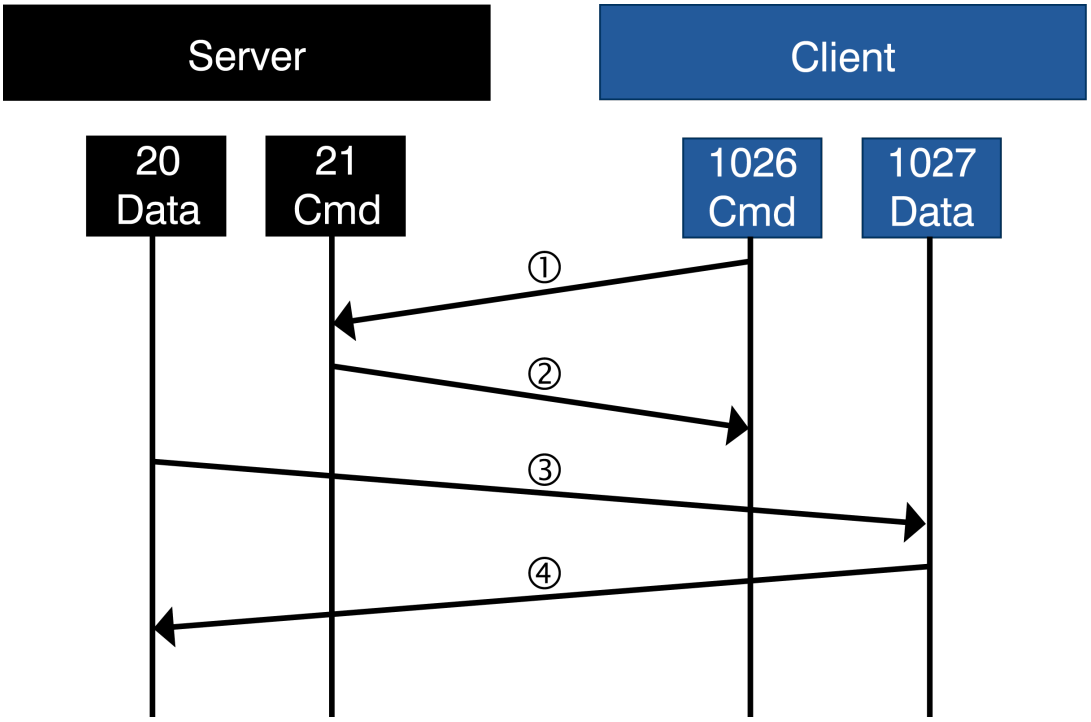
The FTP commands from the client to the server consist of short sets of ASCII characters, followed by optional command parameters. For example, the FTP command to display the current working directory is `PWD` (Print Working Directory). All commands are terminated by a carriage return-linefeed sequence (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.

FTP security could be guarantied by a secured connection (TLS). Server could behave as implicit server: the connection is secured from the start (generally port 990 is used). Or it could behave as an implicit server: the connection is started normally (generally port 21 is used) and updated through the AUTH command.

25.4.1 Active mode FTP

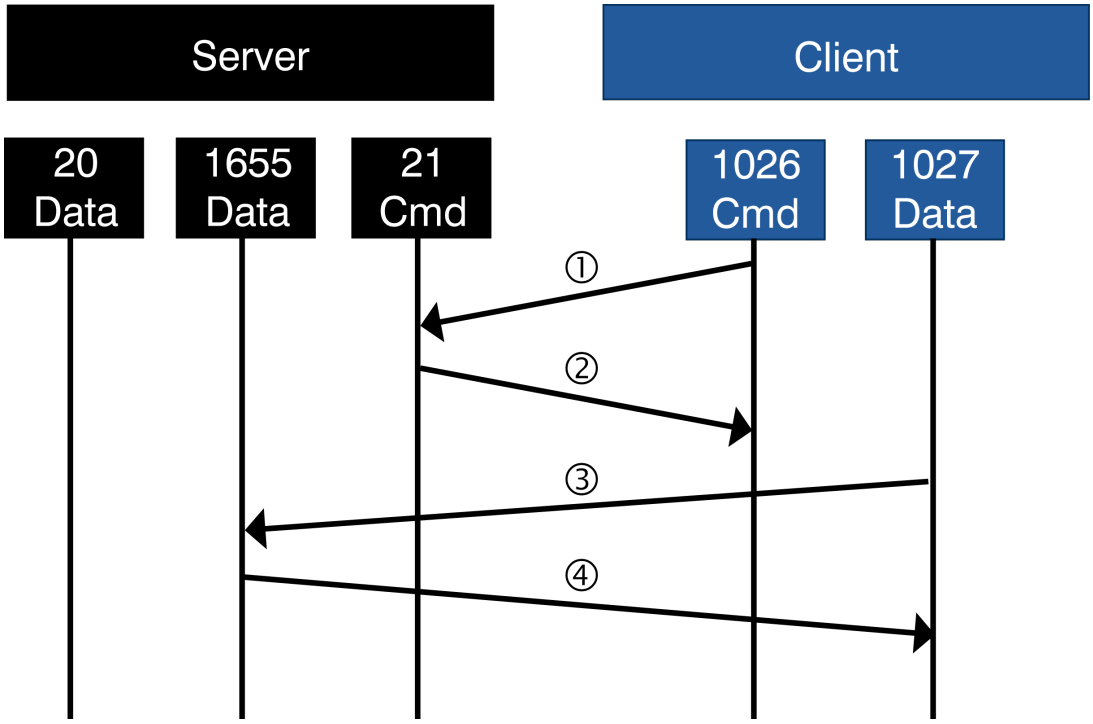
In active mode FTP the client connects from a random unprivileged port P (P > 1023) to the FTP server's command port, port 21. Then, the client starts listening to port P+1 and sends the FTP command PORT P+1 to the FTP server. The server will then connect back to the client's specified data port from its local data port, which is port 20.



25.4.2 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.



25.4.3 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.

25.4.4 Supported FTP commands

emFTP 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	Rename 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
AUTH	Authentication mechanism (TLS)
PBSZ	Protection buffer size
PROT	Data channel protection level.

25.5 Using the emFTP server sample

Ready to use examples for Microsoft Windows and emNet 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 1251 for detailed information.

25.5.1 Using the emFTP server 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 emFTP server. If you do not have the Microsoft compiler, an precompiled executable of the FTP server is also supplied. The base directory of the Windows sample application is `C:\FTP\`.

Building the emFTP server 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`”.

25.5.2 Running the emFTP server example on target hardware

The emFTP server sample application should always be the first step to check the proper function of the emFTP 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.

25.6 Access control

The emFTP server supports a fine-grained access permission scheme. Access permissions can be defined on user-basis for every directory and every file. The access permission of a directory or a file is a combination of the following attributes: visible, readable and writable. To control the access permission four callback functions have be implemented in the user application. The callback functions are defined in the structure FTPS_ACCESS_CONTROL. For detailed information about these structure, refer to *Structure FTPS_ACCESS_CONTROL* on page 730.

25.6.1 pfFindUser()

Description

Callback function which checks if the user is valid.

Prototype

```
int (*pfFindUser)   ( const char * sUser );
```

Parameters

Parameter	Description
sUser	User name.

Return value

= 0 - UserID invalid or unknown
> 0 - UserID, no password required
< 0 - UserID, password required

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;
}
```


25.6.2 pfCheckPass()

Description

Callback function which checks if the password is valid.

Prototype

```
int (*pfCheckPass)  (       int    UserId,
                      const char * sPass );
```

Parameters

Parameter	Description
UserId	Id number
Pass	Password string.

Return value

= 0 - UserID known, password valid.
= 1 - UserID unknown or password invalid

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;
  }
}
```


25.6.3 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 );
```

Parameters

Parameter	Description
UserId	Id number
sDirIn	Directory to check permission for
pDirOut	Directory that can be accessed
SizeOfDirOut	Size of buffer addressed by pDirOut

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.

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 DirOutSize) {
```



```
  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;
}
```


25.6.4 pfGetFileInfo()

Description

Callback function which checks the permissions of the connected user for every file.

Prototype

```
int (*pfGetFileInfo) (       int    UserId,
                       const char * sFileIn,
                             char * pFileOut,
                             int    SizeOfFileOut );
```

Parameters

Parameter	Description
UserId	Id number
sFileIn	File to check permission for
pFileOut	File that can be accessed
SizeOfFileOut	Size of buffer addressed by pFileOut

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.

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 FileOutSize) {
  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;
}
```


25.7 Configuration

The emNet 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).

25.7.1 emFTP server compile time configuration switches

Type	Symbolic name	Default	Description
F	`FTPS_WARN`	--	Defines a function to output warnings. In debug configurations (`DEBUG = 1`) `FTPS_WARN` maps to `IP_Warnf_Application()`.
F	`FTPS_LOG`	--	Defines a function to output logging messages. In debug configurations (`DEBUG = 1`) `FTPS_LOG` maps to `IP_Logf_Application()`.
N	`FTPS_BUFFER_SIZE`	512	Defines the size of the send and receive buffer of the FTP server.
N	`FTPS_MAX_PATH`	128	Defines the maximum length of the buffer used for the path string.
N	`FTPS_MAX_PATH_DIR`	128	Defines the maximum length of the buffer used for the directory string.
N	`FTPS_MAX_FILE_NAME`	13	Defines the maximum length of the buffer used for a filename string.

25.7.2 emFTP server runtime configuration

The input buffer, the output buffer, the path buffer, the directory buffer and the filename buffer are runtime configurable.

Up to version 3.14 the compile time switches `FTPS_AUTH_BUFFER_SIZE`, `FTPS_MAX_PATH`, `FTPS_MAX_PATH_DIR`, `FTPS_MAX_FILE_NAME` and `FTPS_ERR_BUFFER_SIZE` were used to configure the sizes of the buffers. These compile time switches 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_FTPS_ConfigBufSizes()` is not called.

The compile time switches `FTPS_AUTH_BUFFER_SIZE` and `FTPS_ERR_BUFFER_SIZE` are no longer required. For further information, please refer to *`IP_FTPS_ConfigBufSizes`* on page 713.

25.7.3 emFTP server system time

The FTP server requires a system time for the transmission of a complete file timestamp. FTP servers send only a piece of the timestamp of a file, either month, day and year or month, day and time. For the decision which pieces of the timestamp has to be transmitted, it compares the year of the current system time with the year which is stored in the timestamp of the file. Depending on the result of this comparison either the year or the time will be send. The following two examples show the output for both cases.

Example

1. If the value for year in the timestamp of the file is smaller then the value for year in the current system time, year will be sent:
   ```
   -rw-r--r-- 1 root 2000 Jan 1 2007 PAKET00.TXT
   ```
   In this case, the FTP client leaves this column empty or fills the missing time with 00:00. The following screenshot shows the output of the Microsoft Windows command line FTP client:
   ```
   -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 711 for detailed information.

25.7.3.1 pfGetTimeDate()

Description

Returns the current system time.

Prototype

```
int (*pfGetTimeDate) ( void );
```

Return value

Current system time. If no real time clock is implemented, it should return `0x00210000` (1980-01-01 00:00)

Additional information

The format of the time is arranged as follows:
Bit 0-4: 2-second count (0-29)
Bit 5-10: Minutes (0-59)
Bit 11-15: Hours (0-23)
Bit 16-20: Day of month (1-31)
Bit 21-24: Month of year (1-12)
Bit 25-31: Number of years since1980 (0-127)

This function pointer is used in the `FTPS_APPLICATION` structure. Refer to *Structure `FTPS_APPLICATION`* on page 733 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;
}
```


25.8 API functions

Function	Description
IP_FTPS_ConfigBufSizes()	Sets the buffer size used by the FTP server tasks.
IP_FTPS_CountRequiredMem()	Counts the memory required for one thread.
IP_FTPS_Init()	Initializes the application specific FTP server context.
IP_FTPS_Process()	Thread functionality of the FTP server.
IP_FTPS_ProcessEx()	Thread functionality of the FTP server.
IP_FTPS_OnConnectionLimit()	Sends the indication that the connection limit is reached.
IP_FTPS_SetSignOnMsg()	Sets the sign on message for the FTP server.
IP_FTPS_IsDataSecured()	Indicates if the data connection is also secured.
IP_FTPS_AllowOnlySecured()	Makes the server allowing only secured connections in explicit mode (FTPES).
IP_FTPS_SetImplicitMode()	Indicates to the server that implicit mode (FTPS) is active.
IP_FTPS_UseRenameToFullPath()	Makes the server use the full path with the IP_FS layer for a rename operation.
IP_FTPS_SendFormattedString()	Sends a string with placeholders that will be filled using SEGGER_vsnprintfEx() for one line of a sign on message.
IP_FTPS_SendMem()	Sends data via the control connection.
IP_FTPS_SendString()	Sends a zero-terminated string via the control connection.
IP_FTPS_SendUnsigned()	Sends an unsigned value via the control connection.

25.8.1 IP_FTPS_ConfigBufSizes()

Description

Sets the buffer size used by the FTP server tasks.

Prototype

```
void IP_FTPS_ConfigBufSizes(FTPS_BUFFER_SIZES * pBufferSizes);
```

Parameters

Parameter	Description
pBufferSizes	Configuration of buffer sizes.

For detailed information about the structure type FTPS_BUFFER_SIZES refer to *Structure FTPS_BUFFER_SIZES* on page .

25.8.2 IP_FTPS_CountRequiredMem()

Description

Counts the memory required for one thread. This can be used to determine the total required memory pool size for a configuration.

Prototype

```
U32 IP_FTPS_CountRequiredMem(FTPS_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Context keeping track of configured settings. Can be NULL.

Return value

Amount of memory required for internals to handle one thread.

Additional information

In addition to the memory requirement calculated for the FTP server internals, some additional memory might be required for managing a memory pool.

25.8.3 IP_FTPS_Init()

Description

Initializes the application specific FTP server context. This context is specific to one connection. Has to be called if `IP_FTPS_ProcessEx()` is used for the task processing.

Prototype

```
void IP_FTPS_Init(      FTPS_CONTEXT     * pContext,
                  const IP_FTPS_API      * pIP_API,
                  const IP_FS_API        * pFS_API,
                  const FTPS_APPLICATION * pApplication,
                  const FTPS_SYS_API     * pSYS_API);
```

Parameters

Parameter	Description
pContext	Pointer to the FTP server application context. This keeps track of configured settings.
pIP_API	Function table with API functions necessary for IP operations.
pFS_API	Function table with API functions for file system operations.
pApplication	FTP server application settings.
pSYS_API	Function table with API functions necessary for system operations.

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.

For detailed information about the used structure types, please refer to:

Structure	Description
IP_FTPS_API	*Structure IP_FTPS_API* on page 729
IP_FS_API	*File system abstraction layer* on page 1251
FTPS_APPLICATION	*Structure FTPS_APPLICATION* on page 733
FTPS_SYS_API	*Structure FTPS_SYS_API* on page 732

25.8.4 IP_FTPS_Process()

Description

Thread functionality of the FTP server. Initializes and starts the FTP server. Returns when the connection is closed or a fatal error occurs.

Prototype

```
int IP_FTPS_Process(const IP_FTPS_API      * pIP_API,
                          FTPS_SOCKET        hCtrlSock,
                    const IP_FS_API        * pFS_API,
                    const FTPS_APPLICATION * pApplication);
```

Parameters

Parameter	Description
pIP_API	Function table with API functions necessary for IP operations.
hCtrlSock	Handle of the control socket.
pFS_API	Function table with API functions for file system operations.
pApplication	FTP server application settings.

Return value

0 O.K.

Additional information

New implementations should use IP_FTPS_ConfigBufSizes(), IP_FTPS_Init() and IP_FTPS_ProcessEx() instead of IP_FTPS_Process() .

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 hCtrlSock 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 *File system abstraction layer* on page 1251. For detailed information about the structure type FTPS_APPLICATION refer to *Structure FTPS_APPLICATION* on page 733.

25.8.5 IP_FTPS_ProcessEx()

Description

Thread functionality of the FTP server. Returns when the connection is closed or a fatal error occurs.

Prototype

```
int IP_FTPS_ProcessEx(FTPS_CONTEXT * pContext,
                      FTPS_SOCKET    hCtrlSock);
```

Parameters

Parameter	Description
pContext	Pointer to the FTP server application context. This keeps track of configured settings.
hCtrlSock	Handle of the control socket.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

The hCtrlSock is the socket which was created when the client has been connected to the command port (usually port 21).

25.8.6 IP_FTPS_OnConnectionLimit()

Description

Sends the indication that the connection limit is reached.

Prototype

```
void IP_FTPS_OnConnectionLimit(const IP_FTPS_API * pIP_API,
                               FTPS_SOCKET   hCtrlSock);
```

Parameters

Parameter	Description
pIP_API	Function table with API functions necessary for IP operations.
hCtrlSock	Handle of the control socket.

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 hCtrlSock is the socket which was created when the client has been connected to the command port (usually port 21).

25.8.7 IP_FTPS_SetSignOnMsg()

Description

Sets the sign on message for the FTP server.

Prototype

```
void IP_FTPS_SetSignOnMsg(const char * sSignOnMsg);
```

Parameters

Parameter	Description
sSignOnMsg	The “sign on” message.

Additional information

If not set with this function, the default sign on message from the FTPS_SIGN_ON_MSG define will be used.

25.8.8 IP_FTPS_IsDataSecured()

Description

Indicates if the data connection is also secured.

Prototype

```
int IP_FTPS_IsDataSecured(const FTPS_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Pointer to the FTP context.

Return value

0 Data connection is not secured.
1 Data connection is secured.

Additional information

If pContext is NULL, it is assumed that data are secured too.

25.8.9 IP_FTPS_AllowOnlySecured()

Description

Makes the server allowing only secured connections in explicit mode (FTPES). When this API is called, the command connection shall be secured. If the flag `DataOnOff` is set, data connection shall also be secured.

Prototype

```
void IP_FTPS_AllowOnlySecured(FTPS_CONTEXT * pContext,
                              unsigned       DataOnOff);
```

Parameters

Parameter	Description
`pContext`	Pointer to the FTP context.
`DataOnOff`	Flag to indicate if the data connection shall be also secured.

25.8.10 IP_FTPS_SetImplicitMode()

Description

Indicates to the server that implicit mode (FTPS) is active.

Prototype

```
void IP_FTPS_SetImplicitMode(FTPS_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Pointer to the FTP context.

25.8.11 IP_FTPS_UseRenameToFullPath()

Description

Makes the server use the full path with the IP_FS layer for a rename operation.

Prototype

```
void IP_FTPS_UseRenameToFullPath(FTPS_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Pointer to the FTP context.

Additional information

By default many filesystems expect only the new name to be given in a rename operation and might even fail if given a full path. This routine allows the full path of the new name to be given to the IP_FS layer if this is required with your filesystem.

25.8.12 IP_FTPS_SetSignOnMsgCallback()

Description

Sets a callback that gets executed to send a sign on message to a new client.

Prototype

```
void IP_FTPS_SetSignOnMsgCallback(FTPS_CONTEXT                * pContext,
                                  FTPS_SEND_SIGN_ON_MSG_FUNC * pf);
```

Parameters

Parameter	Description
pContext	Pointer to the FTP context.
pf	Callback to set.

Additional information

This API has to be called after `IP_FTPS_Init()`.

Example

```
/*********************************************************************
*
*       _cbSendSignOnMessage()
*
*  Function description
*    Sends a custom sign on message to a client.
*
*  Parameters
*    pOutput: Connection context.
*    Code   : The three digit status code of the message.
*    p      : Reserved for future extensions of this API.
*
*  Additional information
*    A sign on message can consist of multiple lines and has to be
*    in the following format (the value 220 is assumed as Code):
*
*      220-A multi line response starts with the code and a hyphen.\r\n
*      Further lines do not need to use the code in front.\r\n
*      All lines provided by the callback need to end with CRLF.\r\n
*        Lines can start with one or multiple whitespaces.\r\n
*      220 The last line is indicated by the code followed by a whitespace.\r\n
*/
static void _cbSendSignOnMessage(FTPS_OUTPUT* pOutput, unsigned Code, void* p) {
  FTPS_USE_PARA(p);

  //
  // A sign on message can be dynamically generated using a formatted
  // output. The message can also be generated using multiple calls to
  // IP_FTPS_Send* API calls.
  // Please do not forget to add the CRLF at each end of line.
  //
  IP_FTPS_SendFormattedString(pOutput, "%u-Welcome to emFTP server\r\n"
                                       "  This is an example of a multi line sign on message generated via callback.\r\n"
                                       "%u It simply works!\r\n", Code, Code);
}

void main(void) {
  ...
  IP_FTPS_Init(&FTPSContext, &_IP_API, _pFS_API, &_Application, &_Sys_API);
  IP_FTPS_SetSignOnMsgCallback(&FTPSContext, _cbSendSignOnMessage);
  ...
}
```


25.8.13 IP_FTPS_SendFormattedString()

Description

Sends a string with placeholders that will be filled using `SEGGER_vsnprintfEx()` for one line of a sign on message.

Prototype

```
int IP_FTPS_SendFormattedString(      FTPS_OUTPUT * pOutput,
                                const char        * sFormat,
                                                    ...);
```

Parameters

Parameter	Description
`pOutput`	Connection context.
`sFormat`	Formatted string that might contain placeholders.

Return value

Number of characters (without termination) that would have been stored if the buffer had been large enough.

Additional information

Allows sending a string containing placeholders without the need to have the final string created into a temporary buffer in the application. The output buffer is used directly which saves a buffer and avoids unnecessary copy operations from the application to the output buffer.

This routine is only meant to be used from within a callback that has been set using `IP_FTPS_SetSignOnMsgCallback()` .

25.8.14 IP_FTPS_SendMem()

Description

Sends data via the control connection.

Prototype

```
int IP_FTPS_SendMem(      FTPS_OUTPUT * pOutput,
                    const U8          * pData,
                          unsigned      NumBytes);
```

Parameters

Parameter	Description
pOutput	Connection context.
pData	Pointer to a memory location to send.
NumBytes	Number of bytes to send.

Return value

= 0 O.K.
≠ 0 Error

Additional information

This routine is only meant to be used from within a callback that has been set using IP_FTPS_SetSignOnMsgCallback() .

25.8.15 IP_FTPS_SendString()

Description

Sends a zero-terminated string via the control connection.

Prototype

```
int IP_FTPS_SendString(      FTPS_OUTPUT * pOutput,
                       const char        * s);
```

Parameters

Parameter	Description
pOutput	Connection context.
s	String to send.

Return value

= 0 O.K.
≠ 0 Error

Additional information

This routine is only meant to be used from within a callback that has been set using `IP_FTPS_SetSignOnMsgCallback()`.

25.8.16 IP_FTPS_SendUnsigned()

Description

Sends an unsigned value via the control connection.

Prototype

```
int IP_FTPS_SendUnsigned(FTPS_OUTPUT * pOutput,
                         unsigned      v,
                         unsigned      Base,
                         int           NumDigits);
```

Parameters

Parameter	Description
pOutput	Connection context.
v	Value to send.
Base	Numerical base of the value v.
NumDigits	Number of digits to send. 0 can be used as a wildcard.

Return value

= 0 O.K.
≠ 0 Error

Additional information

This routine is only meant to be used from within a callback that has been set using IP_FTPS_SetSignOnMsgCallback() .

25.9 Data structures

25.9.1 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 hSock);
  int         (*pfReceive)   (      unsigned char* pData, int Len,
                              FTPS_SOCKET hSock);
  FTPS_SOCKET (*pfConnect)   (FTPS_SOCKET hCtrlSock, U16 Port);
  void        (*pfDisconnect)(FTPS_SOCKET hDataSock);
  FTPS_SOCKET (*pfListen)    (FTPS_SOCKET hCtrlSock, U16* pPort, U8* pIPAddr);
  int         (*pfAccept)    (FTPS_SOCKET hCtrlSock, FTPS_SOCKET* phDataSocket);
  int         (*pfSetSecure) (FTPS_SOCKET Socket, FTPS_SOCKET Clone);
} 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.
pfSetSecure	Callback function that sets a FTPS_SOCKET (input command connection) as secured and eventually clone it for the output command connection. Could be NULL if TLS security is not supported.

25.9.2 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);
  int (*pfGetFileInfo)(int UserId, const char* sFileIn, char* sFileOut,
                       int SizeOfFileOut);
} 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 permissions of the connected user for every file. May be NULL if directory permissions are sufficient for your needs.

Example

Refer to *Access control* on page 702 for an example.

25.9.3 FTPS_BUFFER_SIZES

Description

Contains the configuration for the buffer to allocate when using `IP_FTPS_ProcessEx()` .

Type definition

```
typedef struct {
  U32  NumBytesInBuf;
  U32  NumBytesInBufBeforeFlush;
  U32  NumBytesOutBuf;
  U32  NumBytesCwdNameBuf;
  U32  NumBytesPathNameBuf;
  U32  NumBytesDirNameBuf;
  U32  NumBytesFileNameBuf;
} FTPS_BUFFER_SIZES;
```

Structure members

Member	Description
`NumBytesInBuf`	Size of Rx buffer. By default `FTPS_BUFFER_SIZE` .
`NumBytesInBufBeforeFlush`	Number of bytes to collect in Rx buffer before they are written to the filesystem. • 0 : Disabled (default). Chunks regardless of their size read will be written directly to the filesystem. • NumBytes: Typically the same as `NumBytesInBuf` and should be a multiple of the block size used by your filesystem e.g. 2k sectors for NAND or SD-card. The data receive function will be called multiple times until the InBuffer gets saturated for the flush or the last chunk of data (connection close) has been read.
`NumBytesOutBuf`	Size of Tx buffer. By default `FTPS_BUFFER_SIZE` .
`NumBytesCwdNameBuf`	Size of buffer used for the Current Working Directory. By default `FTPS_MAX_PATH_DIR` .
`NumBytesPathNameBuf`	Size of buffer used for paths (directory + filename). By default `FTPS_MAX_PATH` .
`NumBytesDirNameBuf`	Size of buffer used for dir(ectory) names (directory without filename). By default `FTPS_MAX_PATH` .
`NumBytesFileNameBuf`	Size of buffer used for filenames. By default `FTPS_MAX_FILE_NAME` .

25.9.4 Structure FTPS_SYS_API

Description

This structure contains the pointers to system functions which are required to use the emFTP server with `IP_FTPS_Init()` and `IP_FTPS_ProcessEx()`.

Prototype

```
typedef struct {
  void* (*pfAlloc)(U32 NumBytesReq);
  void  (*pfFree) (void* p);
} IP_FTPS_API;
```

Member	Description
pfAlloc	Callback function that allocates memory for buffers as configured using `IP_FTPS_ConfigBufSizes()`.
pfFree	Callback function that frees previously allocated resources.

25.9.5 Structure FTPS_APPLICATION

Description

Used to store application specific parameters.

Prototype

```
typedef struct {
  FTPS_ACCESS_CONTROL* pAccess;
  U32 (*pfGetTimeDate) (void);
} FTPS_APPLICATION;
```

Member	Description
pAccess	Pointer to the FTPS_ACCESS_APPLCIATION structure.
pfGetTimeDate	Pointer to the function which returns the current system time.

Additional information

For additional information to structure FTPS_ACCESS_APPLICATION refer to *Structure FTPS_ACCESS_CONTROL* on page 730. For additional information to function pointer pfGetTimeDate() refer to *emFTP server system time* on page 710.

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
};
```


25.9.6 FTPS_SEND_SIGN_ON_MSG_FUNC

Description

Callback executed for sending a sign on message for a new client.

Type definition

```
typedef void (FTPS_SEND_SIGN_ON_MSG_FUNC)(FTPS_OUTPUT * pOutput,
                                           unsigned      Code,
                                           void        * p);
```

Parameters

Parameter	Description
pOutput	Connection context.
Code	The three digit status code of the message.
p	Reserved for future extensions of this API.

Additional information

A sign on message can consist of multiple lines and has to be in the following format (the value 220 is assumed as Code):

220-A multi line response starts with the code and a hyphen.\r\n
Further lines do not need to use the code in front.\r\n
All lines provided by the callback need to end with CRLF.\r\n
  Lines can start with one or multiple whitespaces.\r\n
220 The last line is indicated by the code followed by a whitespace.\r\n

25.10 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 a Cortex-M4 system. Details about the further configuration can be found in the sections of the specific example.

Configuration used

```
#define FTPS_BUFFER_SIZE    512
#define FTPS_MAX_PATH       128
#define FTPS_MAX_PATH_DIR   128
#define FTPS_MAX_FILE_NAME   13
```

25.10.1 ROM usage on a Cortex-M4 system

The following resource usage has been measured on a Cortex-M4 system using SEGGER's Embedded Studio V3.12 using GCC version `6.2.1 20161205 (release) [ARM/embedded-6-branch revision 243739] (arm-none-eabi)` with size optimization.

Addon	ROM
emFTP server	approximately 7.3 kBytes

25.10.2 RAM usage

Using the legacy API `IP_FTPS_Process()`, almost all of the RAM used by the FTP server is taken from task stacks. For new implementations `IP_FTPS_ProcessEx()` should be used, which allocates memory using a callback, keeping the task stack requirement low.

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 including TCP/IP task stack.
ChildTask	Handles a request.	approximately 1800 bytes for the FTP server including TCP/IP stack (excluding filesystem).

Note: The emFTP server requires at least 1 child task.

The approximately RAM usage for the FTP server buffers can be calculated by adding up all buffer sizes configured using `IP_FTPS_ConfigBufSizes()`. The function `IP_FTPS_CountRequiredMem()` can be used to retrieve the required memory for the buffers for one child task for the current configuration.

The FTP server sample is designed to help you to find the correct configuration by using `IP_FTPS_CountRequiredMem()`. In addition the sample warns you about an insufficient memory configuration.

Chapter 26

emFTP client (Add-on)

The emFTP client is an optional extension to the emNet TCP/IP stack. The emFTP client can be used with emNet or with a different TCP/IP stack. All functions which are required to add a emFTP client to your application are described in this chapter.

26.1 emFTP client

The emFTP 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 emFTP client root directory:

Directory	Content
`.\Application\`	Contains the example application to run the FTP client with emNet.
`.\Config\`	Contains the FTP client configuration file.
`.\Inc\`	Contains the required include files.
`.\IP\`	Contains the FTP client sources.
`.\IP\FS\`	Contains the sources for the file system abstraction layer and the read-only file system. Refer to *File system abstraction layer* on page 1251 for detailed information.
`.\Windows\FTPclient\`	Contains the source, the project files and an executable to run emFTP client on a Microsoft Windows host.

26.2 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.

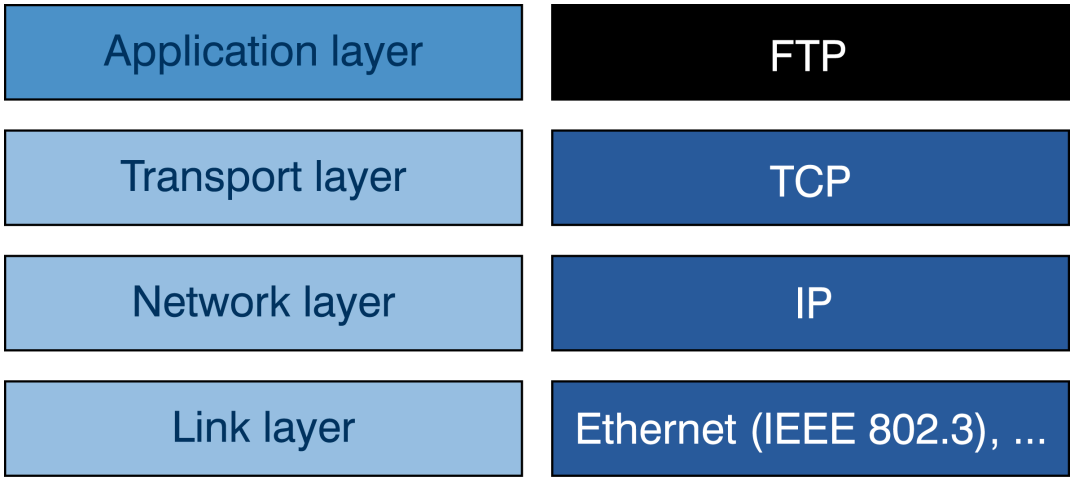
26.3 Requirements

TCP/IP stack

The emFTP client requires a TCP/IP stack. It is optimized for emNet, 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 emNet.

26.4 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.



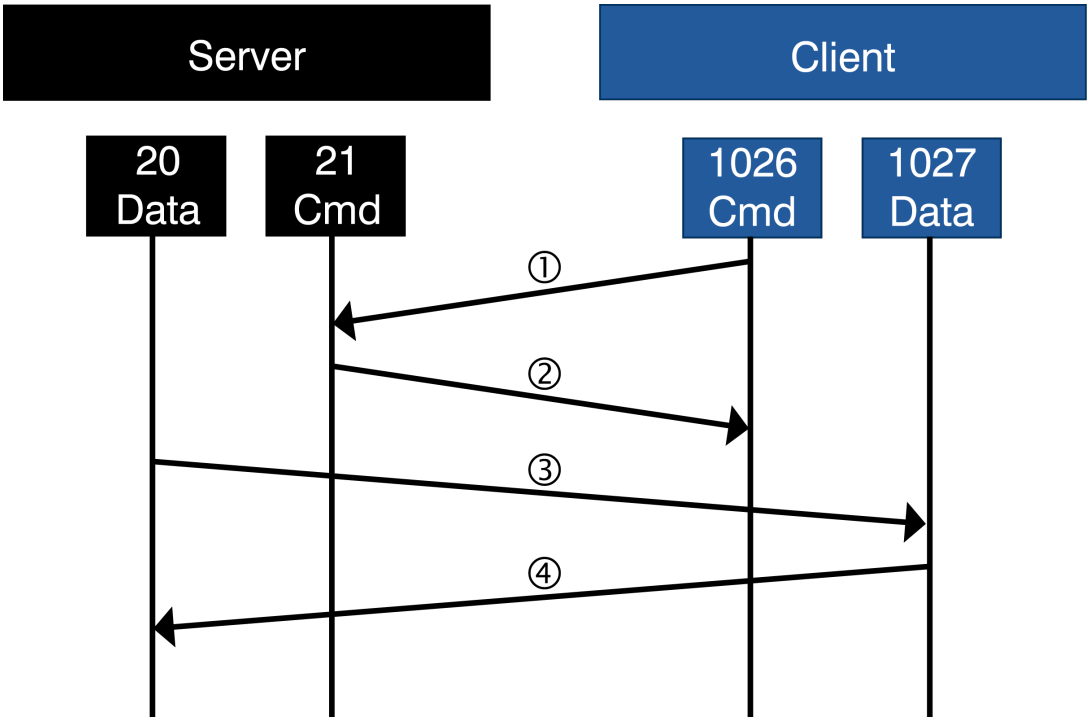
When an FTP client contacts a server, a TCP connection is established between the two machines. The server does a passive open (a socket is listen) when it begins operation; thereafter clients can connect with the server via active opens. This TCP connection persists for as long as the client maintains 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.

26.4.1 Active mode FTP

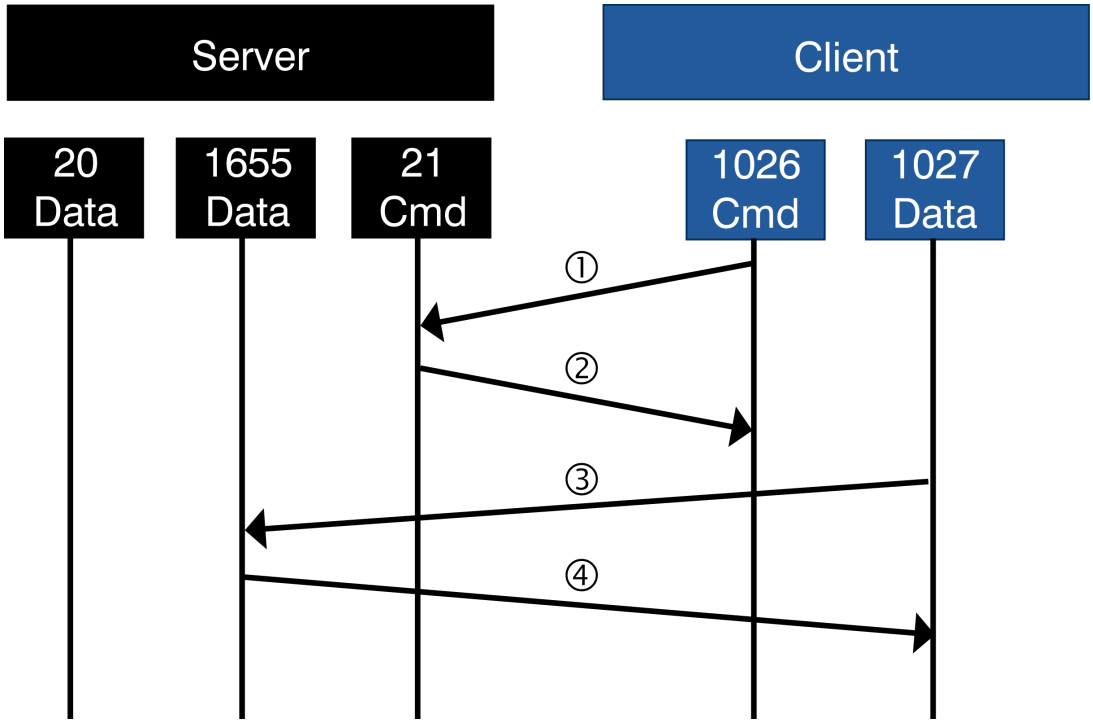
In active mode FTP the client connects from a random unprivileged port P (P > 1023) to the FTP server's command port, port 21. Then, the client starts listening to port P+1 and sends the FTP command PORT P+1 to the FTP server. The server will then connect back to the client's specified data port from its local data port, which is port 20.



26.4.2 Passive mode FTP for the client

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.



26.4.3 Connection security

When a SSL stack is present (for example emSSL), the connection could be secured. To do so the value of the `Mode` parameter of the API `IP_FTPC_Connect()` could be added with `FTPC_MODE_EXPLICIT_TLS_REQUIRED` or `FTPC_MODE_IMPLICIT_TLS_REQUIRED`.

26.4.3.1 FTP implicit mode

When a server works in implicit mode, every connections are secured from the start. It uses a different port than the regular 21 (usually 990). Thus the client has to connect to the right port and upgrade the connection to a secure one.

26.4.3.2 FTP explicit mode

When a server works in explicit mode, the connection start normally as in plain mode. Then the client sends a PROT command to request an upgrade of the connection. When the connection is secured, the exchange goes on with user and password as usual.

26.4.4 Supported FTP client commands

emFTP 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 directory
PASS	Password
PWD	Print the current working directory
RETR	Retrieve
RMD	Remove directory
STOR	Store
APPE	Append
TYPE	Transfer type
USER	User name
PROT	Set protection behavior.
PBSZ	Set protection buffer size.

26.5 Configuration

The emFTP 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).

26.5.1 FTP client compile time configuration switches

Type	Symbolic name	Default	Description
F	FTPc_WARN	--	Defines a function to output warnings. In debug configurations (DEBUG = 1) FTPc_WARN maps to IP_Warnf_Application().
F	FTPc_LOG	--	Defines a function to output logging messages. In debug configurations (DEBUG = 1) FTPc_LOG maps to IP_Logf_Application() .
N	FTPC_BUFFER_SIZE	512	Defines the size of the in and the out buffer of the FTP client. This means that the client requires the defined number of bytes for each buffer. For example, FTPC_BUFFER_SIZE = 512 means 1024 bytes RAM requirement.
N	FTPC_CTRL_BUFFER_SIZE	256	Defines the maximum length of the buffer used for the control channel.
N	FTPC_SERVER_REPLY_BUFFER_SIZE	128	Defines the maximum length of the buffer used for the server reply strings. This buffer is only required and used in debug builds. In release builds the memory will not be allocated.

26.6 API functions

Function	Description
IP_FTPC_Connect()	Connects to a FTP server.
IP_FTPC_Disconnect()	Closes an established connection to a FTP server.
IP_FTPC_ExecCmd()	Executes a FTP command to the FTP server.
IP_FTPC_ExecCmdEx()	Executes a FTP command to the FTP server.
IP_FTPC_Init()	Initializes the context of the FTP client.
IP_FTPC_InitEx()	Initializes the context of the FTP client.

26.6.1 IP_FTPC_Connect()

Description

Connects to a FTP server. The complete login process including the authentication is handled by this function.

Prototype

```
int IP_FTPC_Connect(      IP_FTPC_CONTEXT * pContext,
                    const char            * sServer,
                    const char            * sUser,
                    const char            * sPass,
                          unsigned          PortCmd,
                          unsigned          Mode);
```

Parameters

Parameter	Description
pContext	Pointer to a structure of type IP_FTPC_CONTEXT.
sServer	Dot-decimal IP address of a FTP server, for example “192.168.11.55”.
sUser	User name if required for the authentication. Can be NULL.
sPass	Password if required for the authentication. Can be NULL.
PortCmd	Port number in listening mode on the FTP server. Generally servers are using 21.
Mode	FTP transfer mode. Mode is a mask describing the connection mode and the security. Mask is as follow with default (0) set to FTPC_MODE_ACTIVE and FTPC_MODE_PLAIN_FTP: • FTPC_MODE_ACTIVE or FTPC_MODE_PASSIVE. • FTPC_MODE_PLAIN_FTP or FTPC_MODE_EXPLICIT_TLS_REQUIRED or FTPC_MODE_IMPLICIT_TLS_REQUIRED.

Return value

0	Success.
1	Error. Illegal parameter (pContext = NULL).
-1	Error during the process of connection establishment.

Additional information

The function IP_FTPC_Init() must be called before a call IP_FTPC_Connect().

Note: In the current version of emNet, the FTP client supports only passive mode FTP.

Example

Refer to *IP_FTPC_ExecCmd* on page 749 for an example application which uses IP_FTPC_Connect().

26.6.2 IP_FTPC_Disconnect()

Description

Closes an established connection to a FTP server.

Prototype

```
int IP_FTPC_Disconnect(IP_FTPC_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Pointer to a structure of type IP_FTPC_CONTEXT.

Return value

0 Success.
1 Error. Illegal parameter (pContext = NULL)

Example

Refer to *IP_FTPC_ExecCmd* on page 749 for an example application which uses IP_FTPC_Disconnect().

26.6.3 IP_FTPC_ExecCmd()

Description

Executes a FTP command to the FTP server.

Prototype

```
int IP_FTPC_ExecCmd(      IP_FTPC_CONTEXT * pContext,
                          IP_FTPC_CMD       Cmd,
                    const char            * sPara);
```

Parameters

Parameter	Description
pContext	Pointer to a structure of type IP_FTPC_CONTEXT.
Cmd	The command to perform.
sPara	String with the required parameters for the command. Depending on the command, this parameter can be NULL.

Return value

0	Succes.
1	Error. Illegal parameter (pContext = NULL).
-1	Error during command execution.

Valid values for parameter Cmd

Valid values	Description
FTPC_CMD_CDUP	The command CDUP (Change to Parent Directory). sPara is NULL.
FTPC_CMD_CWD	The command CWD (Change Working Directory). sPara is the path to the directory that should be accessed.
FTPC_CMD_LIST	The command LIST (List current directory content). sPara can be NULL to list the current directory. sPara can be used to specify a directory to list that is not the current working directory.
FTPC_CMD_MKD	The command MKD (Make directory). sPara is the name of the directory that should be created.
FTPC_CMD_PASS	The command PASS (Set password). sPara is the password.
FTPC_CMD_PWD	The command PWD (Print Working Directory). sPara is NULL.
FTPC_CMD_RETR	The command RETR (Retrieve). sPara is the name of the file that should be received from the server. The FTP client creates a file on the used storage medium and stores the retrieved file.
FTPC_CMD_RMD	The command RMD (Remove directory). sPara is the name of the directory that should be removed.

Valid values	Description
FTPC_CMD_STOR	The command STOR (Store). sPara is the name of the file that should be stored on the server. The FTP client opens the file and transmits it to the FTP server.
FTPC_CMD_APPE	The command APPE (Append). sPara is the name of the file that should be appended on the server. The FTP client opens the file and transmits the content to append to the end of the file to the FTP server.
FTPC_CMD_TYPE	The command TYPE (Transfer type). sPara is the transfer type.
FTPC_CMD_USER	The command USER (Set username). sPara is the username.
FTPC_CMD_DELE	The command DELE (delete file). sPara is the name of the file to delete.
FTPC_CMD_PROT	The command PROT (set protection behavior). sPara is the value to set.
FTPC_CMD_PBSZ	The command PBSZ (set protection buffer size). sPara is the value to set (typically 0).

Additional information

IP_FTPC_Init() and IP_FTPC_Connect() have to be called before IP_FTPC_ExecCmd(). Refer to *IP_FTPC_Init* on page 753 for detailed information about how to initialize the FTP client and refer to *IP_FTPC_Connect* on page 747 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 663, 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(&FTPConnection, FTPC_CMD_CWD,  "/Test/");
    if (r == FTPC_ERROR) {
      FTPC_LOG(("APP: Could not change working directory.\r\n"));
      goto Disconnect;
    }
    //
    // Upload the file "Readme.txt
    //
    r = IP_FTPC_ExecCmd(&FTPConnection, FTPC_CMD_STOR, "Readme.txt");
    if (r == FTPC_ERROR) {
      FTPC_LOG(("APP: Could not upload data file.\r\n"));
      goto Disconnect;
    }
    //
    // Change back to root directory.
    //
    r = IP_FTPC_ExecCmd(&FTPConnection, FTPC_CMD_CDUP, NULL);
    if (r == FTPC_ERROR) {
      FTPC_LOG(("APP: Change to parent directory failed.\r\n"));
      goto Disconnect;
    }
    //
    // Disconnect.
    //
Disconnect:
    IP_FTPC_Disconnect(&FTPConnection);
    FTPC_LOG(("APP: Done.\r\n"));
    BSP_ClrLED(0);
    OS_Delay (10000);
  }
}
```


26.6.4 IP_FTPC_ExecCmdEx()

Description

Executes a FTP command to the FTP server.

Prototype

```
int IP_FTPC_ExecCmdEx(IP_FTPC_CONTEXT    * pContext,
                      IP_FTPC_CMD          Cmd,
                      IP_FTPC_CMD_CONFIG * pConfig);
```

Parameters

Parameter	Description
pContext	Pointer to a structure of type IP_FTPC_CONTEXT.
Cmd	The command to perform.
pConfig	Extended configuration of type IP_FTPC_CMD_CONFIG for the command to execute. The old parameter sPara of IP_FTPC_ExecCmd() can also be used via the member IP_FTPC_CMD_CONFIG.sPara .

Return value

0	Succes.
1	Error. Illegal parameter (pContext = NULL).
-1	Error during command execution.

26.6.5 IP_FTPC_Init()

Description

Initializes the context of the FTP client.

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);
```

Parameters

Parameter	Description
pContext	Pointer to a structure of type IP_FTPC_CONTEXT.
pIP_API	Pointer to a structure of type IP_FTPC_API.
pFS_API	Pointer to the filesystem API. Can be NULL if only using commands that do not depend on a filesystem are used. Exmaples for this would be the STOR(e) or APPE(nd) command being used with IP_FTPC_ExecCmdEx() and input from a buffer instead from a file.
pCtrlBuffer	Pointer to the buffer used for the control channel information.
NumBytesCtrl	Size of the control buffer in bytes.
pDataInBuffer	Pointer to the buffer used to receive data from the server.
NumBytesDataIn	Size of the receive buffer in bytes.
pDataOutBuffer	Pointer to the buffer used to transmit data to the server.
NumBytesDataOut	Size of the transmit buffer in bytes.

Return value

0 Success.
1 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 *File system abstraction layer* on page 1251. For detailed information about the structure type IP_FTPC_API refer to *Structure IP_FTPC_API* on page 755.

Example

Refer to *IP_FTPC_ExecCmd* on page 749 for an example application which uses IP_FTPC_Init().

26.6.6 IP_FTPC_InitEx()

Description

Initializes the context of the FTP client.

Prototype

```
int IP_FTPC_InitEx(      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,
                   const IP_FTPC_APPLICATION * pApplication);
```

Parameters

Parameter	Description
pContext	Pointer to a structure of type IP_FTPC_CONTEXT.
pIP_API	Pointer to a structure of type IP_FTPC_API.
pFS_API	Pointer to the filesystem API. Can be NULL if only using commands that do not depend on a filesystem are used. Exmaples for this would be the STOR(e) or APPE(nd) command being used with IP_FTPC_ExecCmdEx() and input from a buffer instead from a file.
pCtrlBuffer	Pointer to the buffer used for the control channel information.
NumBytesCtrl	Size of the control buffer in bytes.
pDataInBuffer	Pointer to the buffer used to receive data from the server.
NumBytesDataIn	Size of the receive buffer in bytes.
pDataOutBuffer	Pointer to the buffer used to transmit data to the server.
NumBytesDataOut	Size of the transmit buffer in bytes.
pApplication	Pointer to a structure of type IP_FTPC_APPLICATION.

Return value

0 Success.
1 Invalid parameters.

26.7 Data structures

26.7.1 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);
  int          (*pfSetSecure)  (FTPC_SOCKET Socket, FTPC_SOCKET Clone);
} 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.
pfSetSecure	Callback function to configure the socket as secured and eventually clone it (Clone might be NULL). This is used when secured connections with SSL are supported. Set this pointer to NULL if no security is present.

26.7.2 Structure IP_FTPC_APPLICATION

Description

This structure contains a callback that will be called for every received lines when getting a reply from the server.

Prototype

```
typedef struct {
  void (*pfReply)(IP_FTPC_CONTEXT* pContext,
                  unsigned         Cmd,
                  const char*      sResponse,
                  unsigned         ResponseLength,
                  unsigned         IsLineComplete);
} IP_FTPC_APPLICATION;
```

Member	Description
pfReply	Callback.
pfReply\pContext	Pointer on the FTP client context.
pfReply\Cmd	Command id.
pfReply\sResponse	Server reply.
pfReply\sResponseLength	Server reply length.
pfReply\IsLineComplete	Last line indication.

26.7.3 IP_FTPC_CMD_CONFIG

Description

Configuration structure used with `IP_FTPC_ExecCmdEx()` for extended functionality.

Type definition

```
typedef struct {
  const char * sPara;
  const char * sLocalPath;
  const char * sRemotePath;
  U8         * pData;
  unsigned     NumBytes;
} IP_FTPC_CMD_CONFIG;
```

Structure members

Member	Description
sPara	Same as sPara with old `IP_FTPC_ExecCmd()` for backwards compatibility. Can be `NULL` if not required for command or sLocalPath and/or sRemotePath are used.
sLocalPath	Local path (terminated string) to use with command. Can be `NULL` if not supported by command (or makes no sense). Overrides sPara . Can be used with the following commands: • FTPC_CMD_STOR • FTPC_CMD_APPE • FTPC_CMD_RETR
sRemotePath	Remote path (terminated string) to use with command. Can be `NULL` if not supported by command (or makes no sense). Overrides sPara . Can be used with the following commands: • FTPC_CMD_LIST • FTPC_CMD_CWD • FTPC_CMD_STOR • FTPC_CMD_APPE • FTPC_CMD_RETR • FTPC_CMD_MKD • FTPC_CMD_RMD • FTPC_CMD_DELE
pData	Indicates that input data shall be taken from the buffer at pData instead of a filesystem. Can be `NULL` if using a filesystem and reading is intended to be done from a filename given by sPara . Can be used with the following commands: • FTPC_CMD_STOR • FTPC_CMD_APPE
NumBytes	Number of bytes to input starting from pData .

26.8 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.
```

26.8.1 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
emFTP client	approximately 2.0 kBytes

26.8.2 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
emFTP client	approximately 1.7 kBytes

26.8.3 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 debug- ging features and disabled debug outputs.	approximately 500 bytes

The approximately task stack size required for the FTP client can be calculated as fol- lows:

TaskStackSize = 2 * `FTPC_BUFFER_SIZE` + `FTPC_CTRL_BUFFER_SIZE`

Build	Description	RAM
Debug	A task used for the FTP client with debugging features and enabled debug outputs.	approximately 500 bytes

The approximately task stack size required for the FTP client can be calculated as follows:

TaskStackSize = 2 * `FTPC_BUFFER_SIZE` + `FTPC_CTRL_BUFFER_SIZE` + `FTPC_SERVER_REPLY_BUFFER_SIZE`

Chapter 27

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 TFPT server to your application are described in this chapter.

27.1 emNet TFTP

The emNet 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.

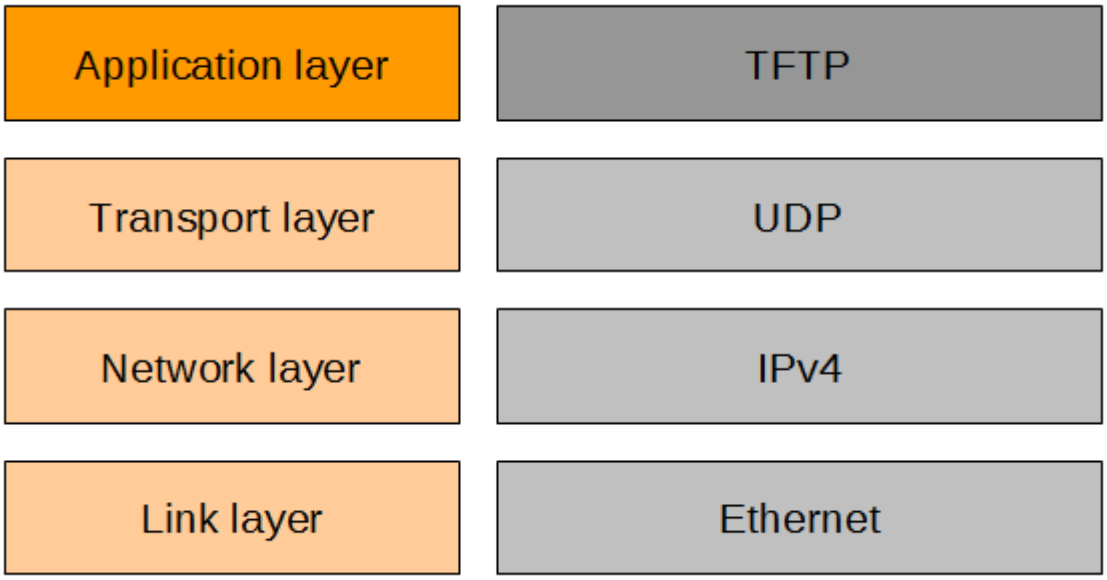
RFC#	Description
[RFC 1350]	TFTP - THE TFTP PROTOCOL (REVISION 2) Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc1350.txt*

27.2 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.

27.3 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.

27.4 Using the TFTP samples

Ready to use examples for emNet 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 868 for detailed information.

27.4.1 Running the TFTP server example on target hardware

The emNet 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.

27.5 API functions

Function	Description
IP_TFTP_InitContext()	Initializes the context for storing connection parameters of a TFTP client/server.
IP_TFTP_RecvFile()	Requests a file from a TFTP server.
IP_TFTP_SendFile()	Sends a data file to a TFTP server.
IP_TFTP_ServerTask()	TFTP server task that can be be started in a separate task.

27.5.1 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);
```

Parameters

Parameter	Description
pContext	Pointer to a structure of type TFTP_CONTEXT.
IFace	Zero-based interface index.
pFS_API	Pointer to the used file system API.
pBuffer	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	Size of buffer assigned with pBuffer.
ServerPort	Port of the server. Can be 0 if the structure is used to connect as a client or if the default TFTP port (69) should be used.

Return value

= 0 Success.
< 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 provide space to store connection parameters.

27.5.2 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);
```

Parameters

Parameter	Description
pContext	Pointer to a structure of type TFTP_CONTEXT.
IFace	Zero-based interface index.
IPAddr	IP addresse of TFTP server.
Port	Port of TFTP server listening.
sFileName	Name of the file to retrieve from server.
Mode	TFTP_MODE_OCTET.

Return value

≥ 0 O.K.
< 0 Error.

Additional information

A static structure of TFTP_CONTEXT needs to initialized with IP_TFTP_InitContext() before using it with this function.

27.5.3 IP_TFTP_SendFile()

Description

Sends a data file to a TFTP server.

Prototype

```
int IP_TFTP_SendFile(      TFTP_CONTEXT * pContext,
                           unsigned       IFace,
                           U32            IPAddr,
                           U16            Port,
                     const char         * sFileName,
                           int            Mode);
```

Parameters

Parameter	Description
pContext	Pointer to a structure of type TFTP_CONTEXT.
IFace	Zero-based interface index.
IPAddr	IP addresse of TFTP server.
Port	Port of TFTP server listening.
sFileName	Name of file to send to server.
Mode	TFTP_MODE_OCTET.

Return value

≥ 0 O.K.
< 0 Error.

Additional information

A static structure of TFTP_CONTEXT needs to initialized with IP_TFTP_InitContext() before using it with this function.

27.5.4 IP_TFTP_ServerTask()

Description

TFTP server task that can be be started in a separate task.

Prototype

```
void IP_TFTP_ServerTask(void * pPara);
```

Parameters

Parameter	Description
pPara	Cast pointer to a structure of type TFTP_CONTEXT.

Additional information

A static structure of TFTP_CONTEXT needs to initialized with IP_TFTP_InitContext() before using it with this function. The task does never return.

27.6 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.

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
emNet TFTP client	approximately 1.2 kBytes
emNet TFTP server	approximately 1.2 kBytes

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
emNet TFTP client	approximately 1.2 kBytes
emNet TFTP server	approximately 1.2 kBytes

27.6.3 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 28

PPP / PPPoE (Add-on)

The emNet implementation of the Point to Point Protocol (PPP) is an optional extension to emNet. 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.

28.1 emNet PPP/PPPoE

The emNet 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 emNet root directory:

Directory	Content
`.\Application\`	Contains the example application to run the PPP implementation with emNet.
`.\Crypto\`	Contains the required files when using MD5 based CHAP authentication.
`.\Inc\`	Contains the required include files.
`.\IP\`	Contains the PPP sources, `IP_PPP.c`, `IP_PPP_CCP.c`, `IP_PPP_CHAP.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.

28.2 Feature list

- Low memory footprint.
- Support PAP authentication protocol
- Support CHAP authentication protocol
- Support for PPP over Ethernet.

28.3 Requirements

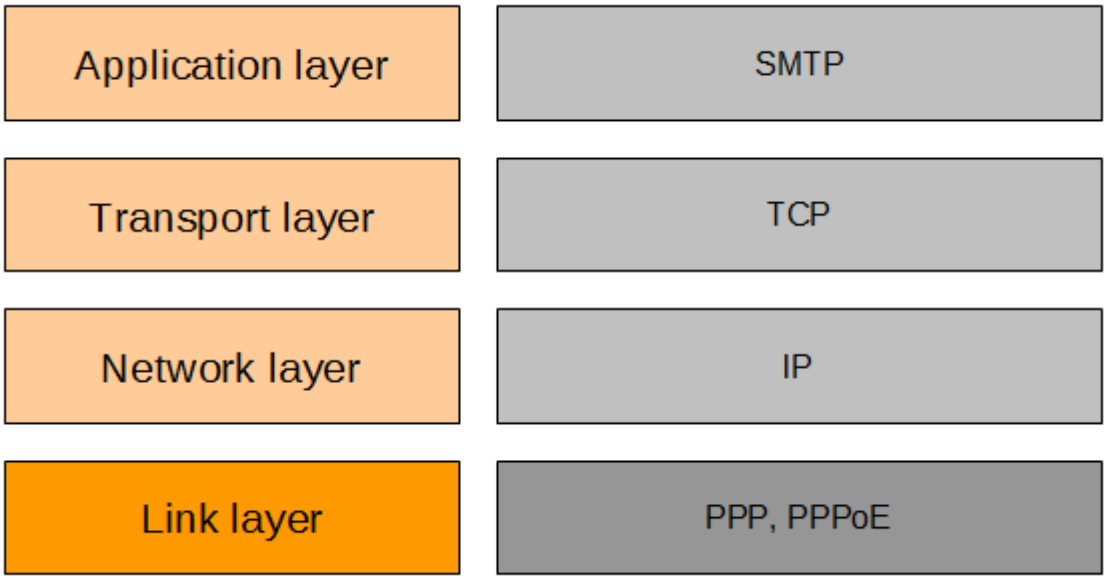
TCP/IP stack

The emNet PPP implementation requires the emNet TCP/IP stack. Your modem has to be able to be configured to respond in the format:

```
"<CR><LF><Response>"
```


28.4 PPP backgrounds

The Point to Point Protocol is a link layer protocol for establishing a direct connection between two network nodes.



Using PPP, an emNet application can establish a PPP connection to a PPP server. The handshaking mechanism includes normally an authentication process. The current version of emNet supports the the following authentication schemes:

- PAP - Password Authentication Protocol
- CHAP - Challenge Handshake Authentication Protocol

28.5 API functions

Function	Description
PPPoE functions	
IP_PPPOE_AddInterface()	Adds a PPPoE interface.
IP_PPPOE_ConfigRetries()	Configures the number of times to resend a lost message before breaking the connection.
IP_PPPOE_Reset()	Resets a PPPoE session.
IP_PPPOE_SetAuthInfo()	Sets the authentication information for the PPPoE connection.
IP_PPPOE_SetUserCallback()	Sets a callback function to inform the user about a status change.
PPP functions	
IP_PPP_AddInterface()	Adds a PPP driver.
IP_PPP_CHAP_AddWithMD5()	Adds support for the CHAP authentication protocol with MD5 algorithm to the stack.
IP_PPP_OnRx()	Receives one or more characters from the hardware.
IP_PPP_OnRxChar()	Receives a character from the hardware (typ.
IP_PPP_OnTxChar()	Sends a character via PPP.
IP_PPP_SetUserCallback()	Sets a callback function to inform the user about a status change.
Modem functions	
IP_MODEM_Connect()	Initializes a PPP connect on a modem using the passed AT command.
IP_MODEM_Disconnect()	Disconnects the connection established with a modem on a specific interface.
IP_MODEM_GetResponse()	Retrieves a pointer to the responses received since the last sent AT command.
IP_MODEM_SendString()	Sends an AT command to the modem without waiting for an answer.
IP_MODEM_SendStringEx()	Sends an AT command to the modem and waits for the expected response with a timeout or checks for responses received in multpiple parts.
IP_MODEM_SetAuthInfo()	Sets authentication information if needed for the connection to establish.
IP_MODEM_SetConnectTimeout()	Sets the connect timeout to wait for a requested connection with IP_MODEM_Connect() to be established.
IP_MODEM_SetInitCallback()	Sets a callback that is used to initialize the modem before actually starting the connection attempt.
IP_MODEM_SetInitString()	Sets an initialization string that is sent to the modem before actually starting the connection attempt.
IP_MODEM_SetUartConfig()	Sets the configuration to be used with the BSP_UART_* API.

Function	Description
IP_MODEM_SetSwitchToCmdDelay()	Sets the delay that is applied before and after the "+++ATH" command that is used to switch back the modem from data to command mode.

28.6 PPPoE functions

28.6.1 IP_PPPOE_AddInterface()

Description

Adds a PPPoE interface.

Prototype

```
int IP_PPPOE_AddInterface(unsigned HWIFaceId);
```

Parameters

Parameter	Description
HWIFaceId	Zero-based interface index to be used as underlying hardware interface.

Return value

≥ 0 Zero-based interface index of the newly created interface.
< 0 Error.

Additional information

Optional configuration of the maximum number of interfaces that can be added to the system using `IP_ConfigMaxIFaces()` needs to be done before adding any interface and must not be changed later.

28.6.2 IP_PPPOE_ConfigRetries()

Description

Configures the number of times to resend a lost message before breaking the connection.

Prototype

```
void IP_PPPOE_ConfigRetries(unsigned IFaceId,
                            U32      NumTries,
                            U32      Timeout);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
NumTries	Number of times the stack will resend the message.
Timeout	Timeout in ms before a resend is triggered.

28.6.3 IP_PPPOE_Reset()

Description

Resets a PPPoE session. 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 IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

28.6.4 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);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
sUser	PPPoE user name.
sPass	PPPoE user password.

28.6.5 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);
```

Parameters

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

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);
```


28.7 PPP functions

28.7.1 IP_PPP_AddInterface()

Description

Adds a PPP driver.

Prototype

```
int IP_PPP_AddInterface(const IP_PPP_LINE_DRIVER * pLineDriver,
                        int                        ModemIndex);
```

Parameters

Parameter	Description
pLineDriver	Pointer to a structure IP_PPP_LINE_DRIVER.
ModemIndex	Modem index; Fixed to 0.

Return value

≥ 0 Zero-based interface index of the newly created interface.
< 0 Error.

Additional information

Optional configuration of the maximum number of interfaces that can be added to the system using IP_ConfigMaxIFaces() needs to be done before adding any interface and must not be changed later.

28.7.2 IP_PPP_CHAP_AddWithMD5()

Description

Adds support for the CHAP authentication protocol with MD5 algorithm to the stack.

Prototype

```
void IP_PPP_CHAP_AddWithMD5(const IP_PPP_MD5_API * pAPI);
```

Parameters

Parameter	Description
pAPI	Pointer to MD5 APi of type IP_PPP_MD5_API .

Additional information

Typically most modern ISPs are happy with the insecure PAP authentication protocol or even suggest it to us directly. In most modern use cases the client is identified for example by its SIM card number and often the ISP does not even care what is sent for username and password. Todays security is typically implemented on higher protocols like TLS on top of TCP. If unsure if required or if you need to be compatible with the one or two providers that absolutely need CHAP these days, you should add this protocol. If not, you should leave it out for a smaller memory footprint.

28.7.3 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(IP_PPP_CONTEXT * pContext,
                 U8             * pData,
                 int              NumBytes);
```

Parameters

Parameter	Description
`pContext`	Pointer to a Structure `IP_PPP_CONTEXT`.
`pData`	Pointer to a buffer which is storing the received data.
`NumBytes`	Number of bytes to receive.

28.7.4 IP_PPP_OnRxChar()

Description

Receives a character from the hardware (typ. modem). Checks if the received character is an escape character, removes the escape character if required and stores the character into packet buffer. When a complete packet is received, it is given to the stack.

Prototype

```
void IP_PPP_OnRxChar(IP_PPP_CONTEXT * pContext,
                     U8               Data);
```

Parameters

Parameter	Description
pContext	Pointer to a structure IP_PPP_CONTEXT.
Data	1 received character.

28.7.5 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 escape character if required.

Prototype

```
int IP_PPP_OnTxChar(unsigned Unit);
```

Parameters

Parameter	Description
Unit	Zero-based interface index.

Return value

0	More data has been sent. Keep Tx interrupt enabled.
1	No more data to send. Disable Tx interrupt if necessary.

28.7.6 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);
```

Parameters

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

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);
```


28.8 Modem functions

28.8.1 IP_MODEM_Connect()

Description

Initializes a PPP connect on a modem using the passed AT command.

Prototype

```
int IP_MODEM_Connect(const char * sATCommand);
```

Parameters

Parameter	Description
sATCommand	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 dial number.

Return value

= 0 Connected
≠ 0 Error

Example

```
IP_MODEM_Connect("ATD*99***1#");
```


28.8.2 IP_MODEM_Disconnect()

Description

Disconnects the connection established with a modem on a specific interface.

Prototype

```
void IP_MODEM_Disconnect(unsigned IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Example

```
IP_MODEM_Disconnect(0);
```


28.8.3 IP_MODEM_GetResponse()

Description

Retrieves a pointer to the responses received since the last sent AT command. It is able to copy the response into a provided buffer if necessary.

Prototype

```
char *IP_MODEM_GetResponse(unsigned   IFaceId,
                           char     * pBuffer,
                           unsigned   NumBytes,
                           unsigned * pNumBytesInBuffer);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pBuffer	Pointer to the buffer where the response shall be copied to. May be NULL.
NumBytes	Size of the buffer pointed to by pBuffer.
pNumBytesInBuffer	Number of bytes in receive buffer. May be NULL.

Return value

= NULL No response in buffer. The last response might have already been cleared to receive the response for the next command.

≠ NULL Pointer to buffer that holds the last response received. Beginning <CR><LF> is skipped.

Example

```
U8 aBuffer[256];
unsigned NumBytesReceived;
IP_MODEM_SendString(0, "AT");
IP_MODEM_GetResponse(0, &aBuffer[0], sizeof(aBuffer), &NumBytesReceived);
```


28.8.4 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);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
sCmd	AT command to be sent.

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()`. If you intend to process the modem response please use `IP_MODEM_SendStringEx()` instead.

28.8.5 IP_MODEM_SendStringEx()

Description

Sends an AT command to the modem and waits for the expected response with a timeout or checks for responses received in multpiple parts.

Prototype

```
int IP_MODEM_SendStringEx(      unsigned   IFaceId,
                          const char     * sCmd,
                          const char     * sResponse,
                                unsigned   Timeout,
                                unsigned   RecvBufOffs);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
sCmd	AT command to be sent. May be NULL.
sResponse	Expected response without <CR><LF> in front. May be NULL.
Timeout	Timeout to wait for any response in ms.
RecvBufOffs	Can be used to check for a response that is sent in multiple parts.

Return value

0	OK, correct response received.
1	Timeout.
2	Wrong response received.

Additional information

Sending a new command with IP_MODEM_SendString() clears the buffer of previous received responses.

RecvBufOffs can be used to check for responses that are sent by the modem in multiple responses. If not passed '0' the receive buffer will not be cleared to not clear out already received following responses from the previously sent command. RecvBufOffs id the offset in bytes from the beginning of the first received response. Being able to receive responses that are sent in multiple parts is necessary as some command may be responded with a confirm for the command sent itself and respond with a second message after an undefined time.

Example sending a command and checking for its response with a timeout

```
IP_MODEM_SendStringEx(0, "AT", "OK", 100, 0);
```

Example for checking the SIM status of a GSM modem

```
int r;

//
// Check if the modem is waiting for a SIM PIN to be entered
//
r = IP_MODEM_SendStringEx(0, "AT+CPIN?\r", "+CPIN: SIM PIN", 1000, 0);
if (r == 0) {
  //
  // The modem is waiting for the PIN to be entered
  //
  IP_MODEM_SendString(0, "AT^SSET=1\r");  // Enable "^SSIM READY" response once
                                          // the SIM data has been read
```



```
  IP_OS_Delay(100);
  //
  // Enter SIM PIN. The OK response will arrive quickly. The modem then
  // reads data from the SIM.
  //
  IP_MODEM_SendStringEx(0, "AT+CPIN="1234"\r", "OK", 15000, 0);
  //
  // After receiving the "OK" response for the command the modem will need an
  // undefined time to read data from the SIM. The modem sends the response
  // "^SSIM READY" once it has finished. We will receive the response at an
  // 6 byte offset (OK<CR><LF><CR><LF>^SSIM READY).
  //
  IP_MODEM_SendStringEx(0, NULL, "^SSIM READY", 15000, 6);
} else {
  //
  // The modem does not seem to wait for a PIN, check if the modem
  // reports "READY". This means no PIN is set for the SIM card. In this case
  // the modem responds with "+CPIN: READY" that will be located at offset 0
  // in the receive buffer.
  //
  if (IP_MEMCMP(IP_MODEM_GetResponse(0, NULL, 0, NULL), "+CPIN: READY", 12) != 0) {
    IP_Panic("Unrecognized response from modem.");
  }
}
```


28.8.6 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);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
sUser	String containing the user name to be used.
sPass	String containing the password to be used.

Additional information

Setting a user name and a password is only necessary when required by your ISP.

Example

```
IP_MODEM_SetAuthInfo(0, "User", "Pass");
```


28.8.7 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);
```

Parameters

Parameter	Description
`IFaceId`	Zero-based interface index.
`ms`	Timeout in `ms`. Default 15s.

Example

```
IP_MODEM_SetConnectTimeout(0, 30000);
```


28.8.8 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)());
```

Parameters

Parameter	Description
`pfInit`	Void callback routine for intialization of the modem before connecting.

Example

```
static void _InitModem(void) {
  IP_MODEM_SendString(0, "AT");
}

IP_MODEM_SetInitCallback(_InitModem);
IP_MODEM_Connect("ATD*99***1#");
```


28.8.9 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);
```

Parameters

Parameter	Description
`sInit`	Command to be sent to the modem before connecting.

Example

```
IP_MODEM_SetInitString("ATE0V1");
IP_MODEM_Connect("ATD*99***1#");
```


28.8.10 IP_MODEM_SetUartConfig()

Description

Sets the configuration to be used with the BSP_UART_* API.

Prototype

```
void IP_MODEM_SetUartConfig(unsigned int  Unit,
                            unsigned long Baudrate,
                            unsigned char NumDataBits,
                            unsigned char Parity,
                            unsigned char NumStopBits);
```

Parameters

Parameter	Description
Unit	Index of UART unit to use.
Baudrate	Baudrate [Hz] to use.
NumDataBits	Number of data bits to use.
Parity	Parity of type BSP_UART_PARITY_* .
NumStopBits	Number of stop bits to use.

Additional information

By default the IP_MODEM_* API calls BSP_UART_Init() with all parameters like unit and baudrate set to zero as these parameters were originally not present. When they eventually became available, zero parameters meant using the modules default values. Unfortunately the default values are not loaded by all BSP_UART implementation or a different unit than unit #0 is planned to be used. To avoid changing the complete API of IP_MODEM module, the parameters can now be set using this API.

BSP_UART_Init() previously was also called with the interface id as unit number. This is now also changed to use either zero for default or the configured value from this routine.

28.8.11 IP_MODEM_SetSwitchToCmdDelay()

Description

Sets the delay that is applied before and after the "+++ATH" command that is used to switch back the modem from data to command mode.

Prototype

```
void IP_MODEM_SetSwitchToCmdDelay(unsigned IFaceId,
                                  unsigned ms);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
ms	Timeout in ms between sending "+++" and "ATH".

Additional information

Sending "+++ATH" to switch back to command mode and then hanging up the connection is fine to be sent 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.

28.9 Data structures

Structure	Description
IP_PPP_CONTEXT	Structure which stores the information about the PPP connection.
RESEND_INFO	A structure which stores the resend condition for different stages of the PPP connection.
IP_PPP_LINE_DRIVER	Structure with pointers to application related functions.

28.9.1 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.

Member	Description
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 initialize 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.

Member	Description
Auth.Prot	Defines the PPP authentication protocol, is set typically to PPP_PROT_PAP.
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

28.9.2 Structure RESEND_INFO

Description

A structure which stores the resend condition for different stages of the PPP connection.

Prototype

```
typedef struct {
  IP_PACKET * pPacket;
  I32         Timeout;
  I32         InitialTimeout;
  U32         RemTries;
#if IP_DEBUG
  const char * sPacketName;
#endif
} RESEND_INFO;
```

Member	Description
pPacket	Pointer to an IP_PACKET structure.
Timeout	Timeout in ms before a resend is triggered.
InitialTimeout	Initial timeout in ms before a resend is triggered. Saved to be able to reset Timeout to it's original state.
RemTries	Counter for the remaining number of retries.
sPacketName	(Only with IP_DEBUG ≥ 1.) Custom name assigned to the packet.

28.9.3 Structure IP_PPP_LINE_DRIVER

Description

Structure with pointers to application related functions.

Prototype

```
typedef struct {
  void (*pfInit) (struct IP_PPP_CONTEXT * pPPPContext);
  void (*pfSend) (U8 Data);
  void (*pfSendNext) (U8 Data);
  void (*pfTerminate) (U8 IFaceId);
  void (*pfOnPacketCompletion) (void);
} IP_PPP_LINE_DRIVER;
```

Member	Description
pfInit	Pointer to a function which initializes 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.

28.10 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.

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
emNet PPP used for PPPoE	approximately 7.0 kBytes

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
emNet PPP used for PPPoE	approximately 6.5 kBytes

28.10.3 RAM usage

Addon	RAM
emNet PPP used for PPPoE	approximately 100 Bytes

28.11 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.

28.11.1 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
emNet PPP used for PPPoE	approximately 7.0 kBytes

28.11.2 RAM usage

Addon	RAM
emNet PPP used for PPPoE	approximately 0.5 kBytes

Chapter 29

NetBIOS (Add-on)

The emNet implementation of the Network Basic Input/Output System Protocol (NetBIOS) is an optional extension to emNet. 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.

29.1 emNet NetBIOS

The emNet 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 emNet root directory:

Directory	Content
`.\Application\`	Contains the example application to run the NetBIOS implementation with emNet.
`.\Inc\`	Contains the required include files.
`.\IP\`	Contains the NetBIOS sources `IP_Netbios.c`.

29.2 Feature list

- Low memory footprint.
- Seamless integration with the emNet stack.
- Client based NetBIOS name resolution.

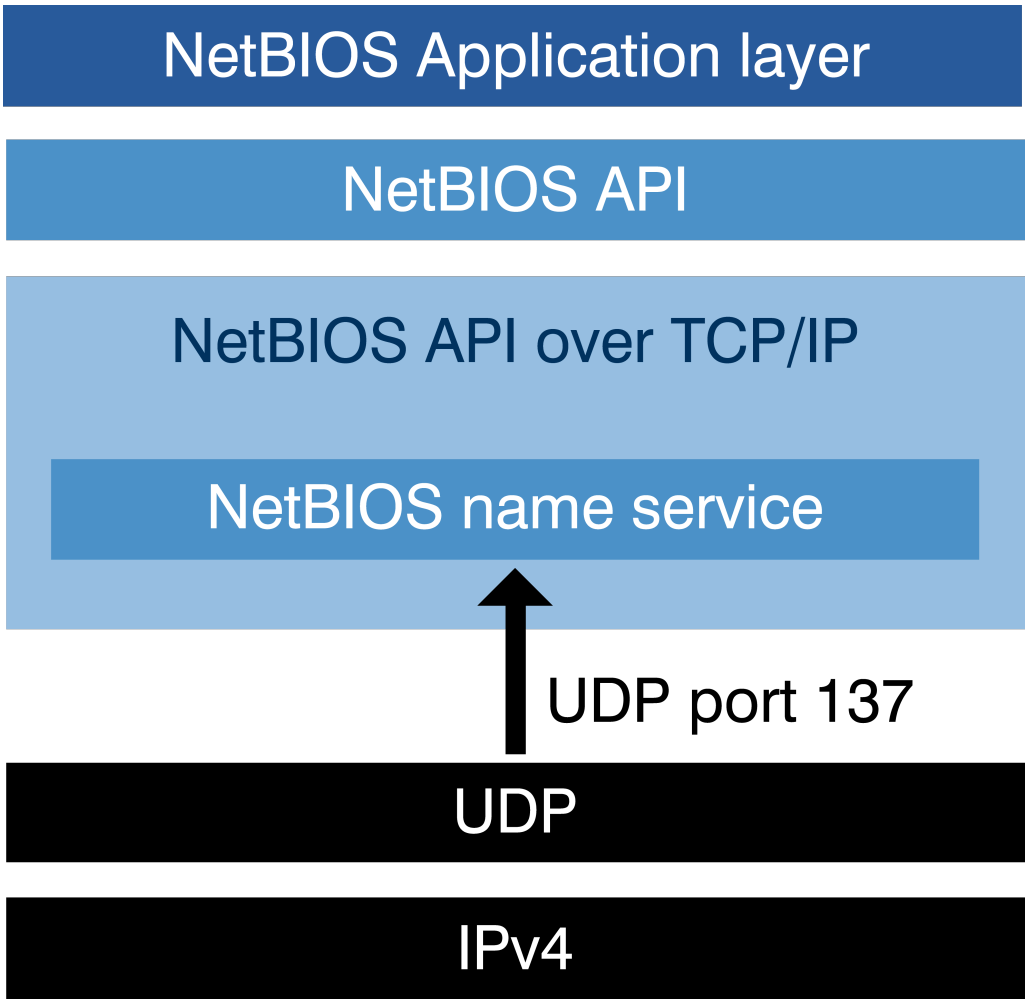
29.3 Requirements

TCP/IP stack

The emNet NetBIOS implementation requires the emNet TCP/IP stack.

29.4 NetBIOS backgrounds

The Network Basic Input/Output System protocol is an API on top of the TCP/IP protocol, it provides a way of communication between separate computers within a local arena network via the session layer.



Using NetBIOS, an emNet application can resolve a NetBIOS name to an IP adress in the local area network.

29.5 API functions

Function	Description
NetBIOS	
IP_NETBIOS_Init()	Initializes the NetBIOS Name Service client.
IP_NETBIOS_Start()	Starts the NetBIOS client Creates an UDP socket to receive NetBIOS Name Service requests.
IP_NETBIOS_Stop()	Stops the NetBIOS client Closes the UDP socket.

29.5.1 IP_NETBIOS_Init()

Description

Initializes the NetBIOS Name Service client.

Prototype

```
int IP_NETBIOS_Init(      U32                IFaceId,
                    const IP_NETBIOS_NAME * paHostnames,
                          U16                LPort);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
paHostnames	Pointer to an array of Structure IP_NETBIOS_NAME. Expects last index to be filled with zero.
LPort	Local port used for listening. Typically 137. If parameter LPort is 0, 137 will be used.

Return value

< 0 Error, invalid NetBIOS name.
> 0 Ok, Number of valid NetBIOS names assigned to the target.

29.5.2 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);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

Return value

= 0 Error, could not create an UDP socket.
> 0 OK, number of the socket which is used for the NetBIOS Name Service.

29.5.3 IP_NETBIOS_Stop()

Description

Stops the NetBIOS client Closes the UDP socket.

Prototype

```
void IP_NETBIOS_Stop(U32 IFaceId);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.

29.5.4 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	Pointer to a string which stores the NetBIOS name.
NumBytes	Length of the NetBIOS name without termination.

29.6 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.

29.6.1 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
emNet NetBIOS module	approximately 0.8 kBytes

29.6.2 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
emNet NetBIOS module	approximately 0.7 kBytes

29.6.3 RAM usage

Addon	RAM
emNet NetBIOS module	approximately 26 bytes

Chapter 30

SNTP client (Add-on)

The emNet implementation of the Simple Network Time Protocol (SNTP) client is an optional extension to emNet. 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.

30.1 emNet SNTP client

The emNet 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 emNet SNTP client root directory:

Directory	Content
`.\IP\`	Contains the SNTPc sources `IP_SNTPC.c`.

30.2 Feature list

- Low memory footprint.
- Seamless integration with the emNet stack.
- Time synchronization with a remote NTP server.

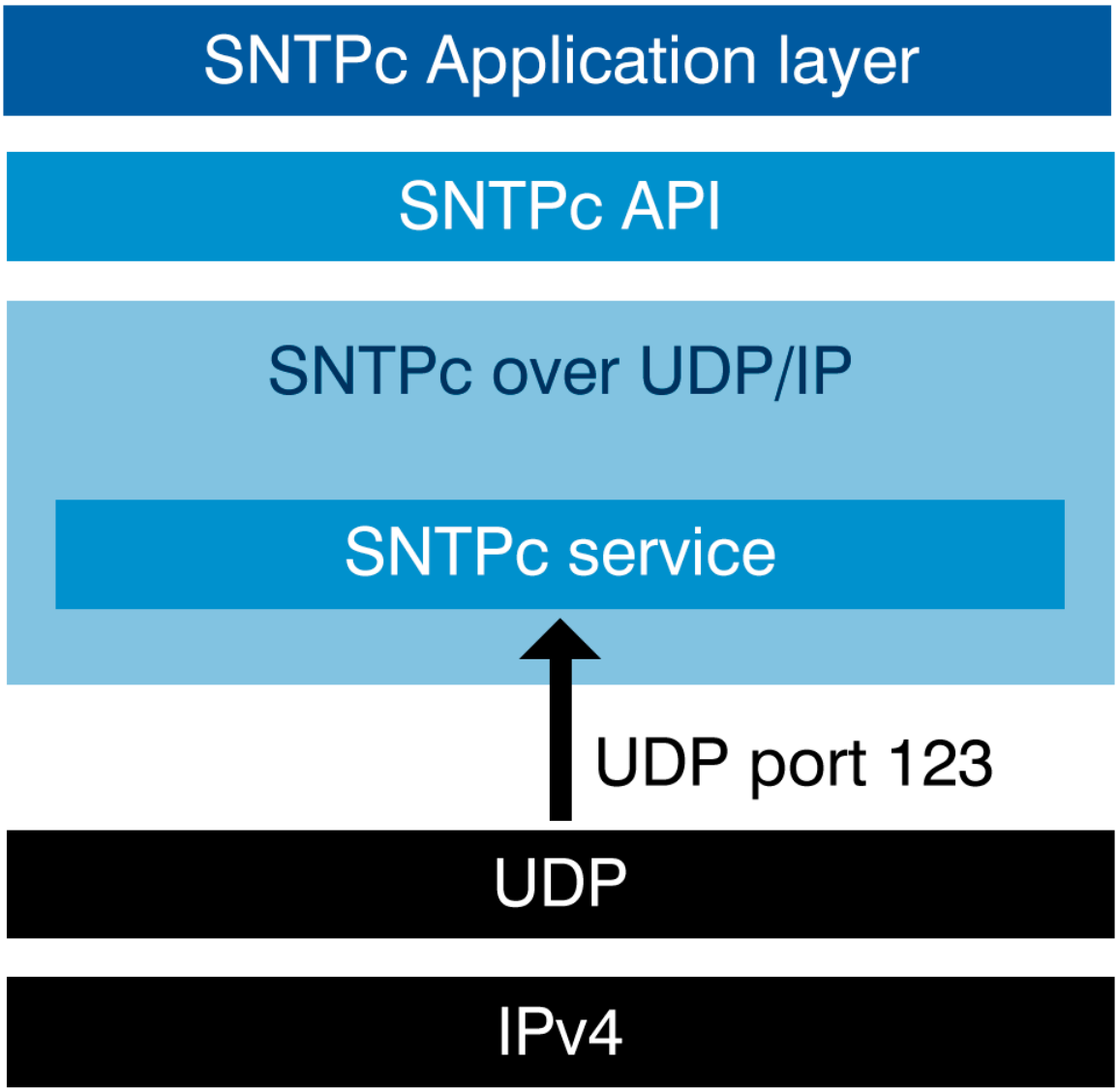
30.3 Requirements

TCP/IP stack

The emNet SNTPc implementation requires the emNet TCP/IP stack.

30.4 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 emNet 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.

30.4.1 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*.

30.4.2 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.

30.5 API functions

Function	Description
SNTP client	
IP_SNTPC_ConfigAcceptNoSyncSource()	Configures if a timestamp from that indicates it originates from a source that is not synchronized is accepted.
IP_SNTPC_ConfigTimeout()	Configures the maximum time to wait for a response from a NTP server.
IP_SNTPC_GetTimeStampFromServer()	Requests the actual time from a NTP server.

30.5.1 IP_SNTPC_ConfigAcceptNoSyncSource()

Description

Configures if a timestamp from that indicates it originates from a source that is not synchronized is accepted.

Prototype

```
void IP_SNTPC_ConfigAcceptNoSyncSource(U8 OnOff);
```

Parameters

Parameter	Description
OnOff	• = 0: Disabled (default). • = 1: Enabled, accept all timestamps sources.

30.5.2 IP_SNTPC_ConfigTimeout()

Description

Configures the maximum time to wait for a response from a NTP server

Prototype

```
void IP_SNTPC_ConfigTimeout(unsigned ms);
```

Parameters

Parameter	Description
ms	Timeout in ms.

30.5.3 IP_SNTPC_GetTimeStampFromServer()

Description

Requests the actual time from a NTP server. Server is passed via punctual IP addresse or DNS name.

Prototype

```
int IP_SNTPC_GetTimeStampFromServer(      unsigned           IFaceId,
                                    const char             * sServer,
                                          IP_NTP_TIMESTAMP * pTimestamp);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
sServer	String containing either dotted decimal IP address (129.250.35.251) or DNS name (us.pool.ntp.org) of NTP server.
pTimestamp	Pointer where to store the received timestamp.

Return value

= 0 IP_SNTPC_STATE_NO_ANSWER, Request sent but no answer from server received within timeout.
= 1 IP_SNTPC_STATE_UPDATED, Timestamp updated from server response.
= 2 IP_SNTPC_STATE_KOD, Server sent Kiss-Of-Death and wants us to use another server.
< 0 Other, Error

30.5.4 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.

30.6 Resource usage

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

30.6.1 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
emNet SNTP client	approximately 0.5 kBytes

30.6.2 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
emNet SNTP client	approximately 0.5 kBytes

30.6.3 RAM usage

Addon	RAM
emNet SNTP client	approximately 24 Bytes

Chapter 31

PTP Ordinary Clock (Add-on)

The emNet implementation of the Precision Time Protocol (PTP) is an optional extension to emNet. Its primary purpose is to synchronize an embedded system to a master clock in a local network. In addition a simple master functionality can be provided for other PTP slaves in case no better master is available in the network. All functions that are required to add PTP functionality to your application are described in this chapter.

31.1 emNet PTP OC

The emNet PTP implementation is an optional extension which can be seamlessly integrated into your application. It combines a maximum of performance with a small memory footprint. This implementation covers PTP version 2, primarily for ordinary slave-only clocks with the Delay/Response mechanism. Only one PTP interface is supported at this time.

The PTP interface can be used as a slave, master or starting as slave/master with the master being disabled later on when a master with a better clock source enters the network.

The PTP module implements the mandatory parts of the following IEEE Standard.

IEEE-Std	Description
[1588-2008]	IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. A copy could be ordered here: *http://www.nist.gov/el/isd/ieee/ieee1588.cfm*

The following table shows the contents of the emNet root directory:

Directory	Content
`.\IP\`	Contains the PTP Ordinary Clock add-on sources. If available and purchased, a driver for hardware timestamps for a specific network interface `IP_DRIVER_NAME_PTP.c` might be included as well.

31.2 emNet PTP OC slave

The PTPv2 protocol allows an embedded system to have a precise synchronization with a master clock present in its local network. This master clock periodically sends timing related messages. The slave can use the received timestamp together with other synchronization messages to the master to calculate an accurate offset to the timestamp that is periodically sent by the master.

31.3 emNet PTP OC master

The emNet PTP implementation primarily targets the PTP slave role as this is the typical role that will be assigned to an embedded target. Most embedded targets do not qualify for a good PTP master as they are not able to provide a clock that qualifies as stable enough for a serious PTP clock source compared to commercial PTP grandmaster clocks that typically sync to a high precision time source like GPS.

emNet therefore only provides a “Simple PTP Master”. While it can be used to provide any PTP master for a network at all, it should not be considered a serious master clock source for precision above multiple microseconds, of course under the assumption that the master can provide a timestamp that accurate at all.

31.4 Hardware timestamp support

By default the emNet PTP implementation uses software timestamps that rely on the precision of the system tick, typically provided by an RTOS via the OS abstraction layer. This tick source typically provides around 1 millisecond precision. On top of the timer precision the code run-time to collect the timestamp and assign it to a specific packet has to be taken into account. The precision might be further downgraded by interrupts and modern CPU features such as pipelining and caches that make a precise code run-time unpredictable.

Some hardwares support PTP timestamps to be taken automatically based on a dedicated PTP timer unit that then is capable of providing a much better and constant time source compared to software timestamps. Hardware timestamps can easily reach a precision in the order of nanoseconds if the PTP hardware timer allows it. To utilize the PTP timer unit, a PTP driver suitable for the specific device and/or Ethernet controller needs to be added.

Warning

Software and hardware timestamping can not be combined due to API restrictions. Typically this is also not necessary as master and slave timestamping works in the same way and can therefore be used for both purposes, even at the same time.

Typically only the timestamps are provided by hardware with the actual PTP logic remaining in software in the stack. External PTP units like the Renesas EtherC PTP unit provide their own PTP logic that does more than just provide timestamps and can not be easily combined with a software logic. Therefore it is currently not possible combine EtherC PTP hardware timestamp for slave purposes with a software master logic.

31.5 Feature list

- Low memory footprint.
- Seamless integration with the emNet stack.
- Time synchronization with a remote PTP master clock.
- Providing a “Simple PTP Master” for the network.

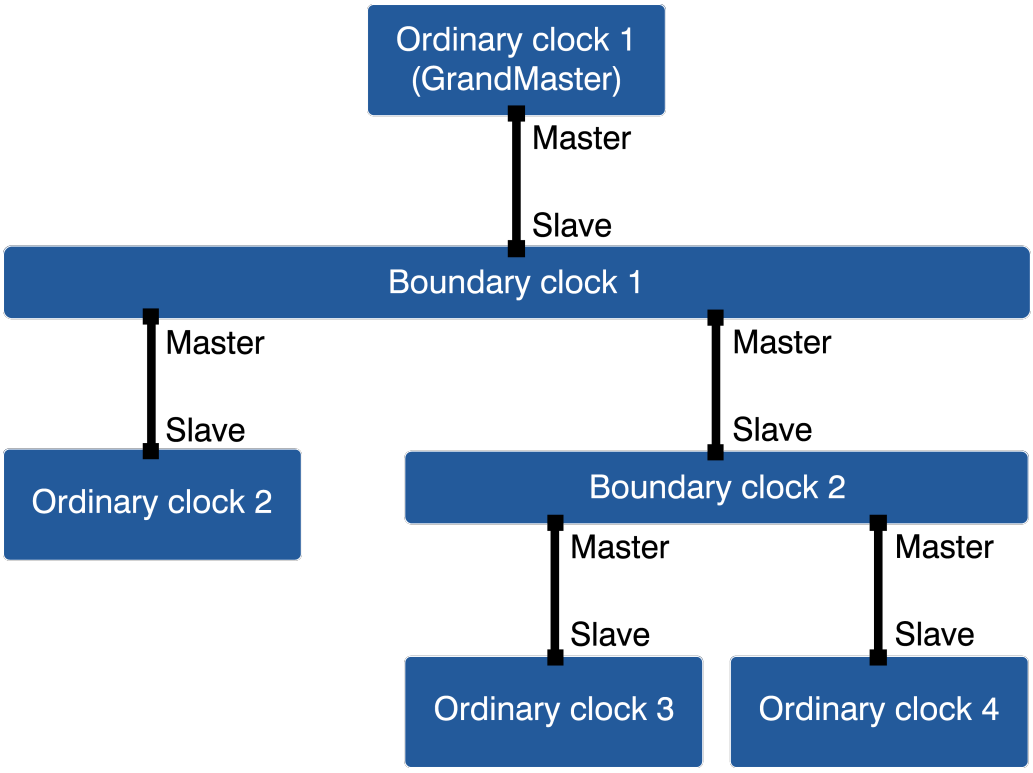
31.6 Requirements

TCP/IP stack

The emNet PTP implementation requires the emNet TCP/IP stack. Although it is working with a complete software solution, a hardware supporting PTP and an associated driver is needed to reach a higher precision.

31.7 PTP background

The PTP protocol is an API on top of the UDP/IP protocol or plain Ethernet. It provides a way of synchronizing the target with a master clock over the local network.



PTP can be used with different protocols. This implementation currently supports PTP over UDP IPv4, PTP over Ethernet and PTP over UDP IPv6 (If IPv6 add-on is present). PTP messages are sent using multicast messages:

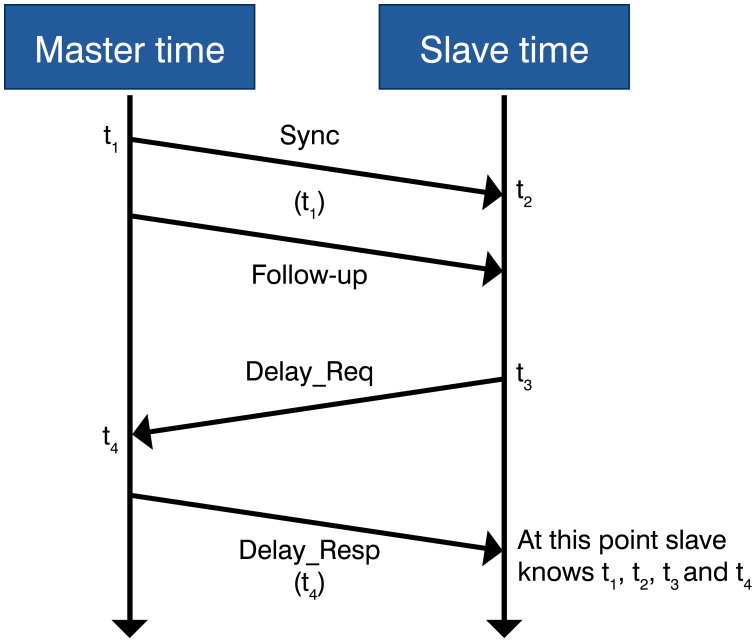
- Over UDP IPv4: mutilcast IP address is 224.0.1.129.
- Over Ethernet: multicast MAC address is 01:1B:19:00:00:00.
- Over UDP IPv6: multicast IP address is FF0X::181 (In this implementation X is set to 2 which corresponds to link-local multicast).

In case more than one master clock is present in the local network, a decision needs to be done which clock is the best master clock. This decision is done by comparing the parameters periodically sent by every master clock. The best master clock is then used as the reference called the GrandMaster clock.

Parameters that define a clock (like priority, description, …) can be obtained (and for some parameters modified) using "Management" messages. See the IEEE1588-2008 specification for more details. (The emNet PTP implementation currently does not support "Management" messages)

Using PTP, an emNet system can synchronize its time with a PTP master clock in its local network by obtaining the transmission/reception timestamp of a series of messages between the master and the slave. With these timestamps, the slave is able to compute the time offset and the propagation time between both clocks.

This series of messages for the Delay/Response mechanism occurs periodically in order to correct the deviation of the slave time with respect to the master.



- At t_1 the master clock sends a Sync message. It either contains t_1 or it is followed by a Follow-Up with t_1 value.
- The slave stores at which time t_2 it receives the Sync message.
- The slave sends a `Delay_Req` and stores at which time t_3 the message is sent.
- The master sends to the slave in a `Delay_Resp` at which time t_4 it has received the `Delay_Req`.
- At the end, the slave has t_1, t_2, t_3 and t_4 and can compute the time offset.

Note that this implementation is not compatible with PTP version 1. Peer-to-peer synchronization mechanism is also not supported. And as previously stated this implementation is for slave-only ordinary clocks. Similarly the specification defines optional features that are currently not supported.

31.7.1 Time representation

The PTP timestamp is represented by a structure of two 32-bit fields. The first 32-bit field contains the complete seconds passed since the EPOCH. The other 32-bit field contains the sub-second part of the time passed since the EPOCH expressed in nanoseconds.

The EPOCH of a PTP system could be any reference used by the grandmaster clock of the network, but generally GPS TAI is used as reference with an EPOCH set to January the 1st 1970. The `Temps Atomique International (TAI)`, the international atomic time is currently ahead of UTC by 37 seconds and always ahead of GPS by 19 seconds.

A nice live view of different time representations live can be found at the following location: http://leapsecond.com/java/gpsclock.htm

31.7.2 Hardware support

The emNet PTP add-on is able to work without the support of hardware. In this case is uses a global IP callback that retrieves the time in microseconds. A default callback is provided that uses the internal milliseconds time. It is possible to provide a more precise function with `IP_SetMicrosecondsCallback()`.

If the hardware supports PTP, a PTP driver can be added during the configuration with `IP_NI_AddPTPDriver()`. Hardware based timestamps are of course more precise than any software timestamp.

The structure `IP_PACKET` (accessible using the zero-copy interface) contains the timestamp of arrival in seconds and nanoseconds for received packets (or time in microseconds when PTP is not available in the driver).

31.8 PTP configuration

The emNet PTP ordinary clock 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.

31.8.1 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).

31.8.2 Configuration switches

Type	Symbolic name	Default	Description
B	`IP_SUPPORT_PTP`	0	Activates the support of PTP in the stack, in the driver and in the PTP module itself.

31.9 API functions

Function	Description
PTP functionality	
IP_PTP_GetDefaultDsClockIdentity()	Retrieves the DefaultDS (DataSet) ClockIdenitity.
IP_PTP_GetTime()	Retrieves the current PTP time based on the HW (driver) or based on the OS time.
IP_PTP_Halt()	Halts the PTP service based on a configuration context.
IP_PTP_Init()	Initializes a PTP context.
IP_PTP_SetTime()	Sets the PTP time (by applying a positive coarse correction if necessary) and resets the PTP sync state to uncalibrated state.
IP_PTP_Start()	Starts the PTP service based on a configuration context.
IP_PTP_OC_AddMasterFallbackLogic()	Adds a fallback PTP OC logic that is used by drivers that do not come with their own logic in hardware.
IP_PTP_OC_AddSlaveFallbackLogic()	Adds a fallback PTP OC logic that is used by drivers that do not come with their own logic in hardware.
IP_PTP_MASTER_Add()	Adds a PTP master to an interface.
IP_PTP_MASTER_Config()	Configures parameters for a PTP Master.
IP_PTP_MASTER_Remove()	Removes a PTP master including fallback logic assigned to it and removes it from its assigned IP_PTP_CONTEXT as well.
IP_PTP_SLAVE_Add()	Adds a PTP slave to an interface.
Optional PTP driver support	
IP_NI_AddPTPDriver()	Adds an NI specific PTP driver for HW timestamp support.
Optional configuration	
IP_PTP_OC_SetInfoCallback()	Sets a callback that gets notified on updates of the PTP clock.
IP_PTP_OC_SetProductDescription()	Sets the product description of the Ordinary Clock.
IP_PTP_OC_SetUserDescription()	Sets the user description of the Ordinary Clock.
IP_PTP_OC_SetRevision()	Sets the revision of the Ordinary Clock.
Old API	
IP_PTP_OC_Halt()	Halts the PTP messages processing on the previously started interface.
IP_PTP_OC_Start()	

31.9.1 IP_PTP_GetDefaultDsClockIdentity()

Description

Retrieves the DefaultDS (DataSet) ClockIdenitity.

Prototype

```
U8 *IP_PTP_GetDefaultDsClockIdentity(IP_PTP_CONTEXT * pContext,
                                     U8             * pBuffer,
                                     unsigned         NumBytes);
```

Parameters

Parameter	Description
pContext	Pointer to PTP context.
pBuffer	Pointer to buffer where to store up to NumBytes of the DefaultDS ClockIdentity. Zero bytes might be stored if PTP has not yet been initialized and/or started. Can be NULL.
NumBytes	Number of bytes of the ClockIdentity to store at pBuffer . Ignored if pBuffer is NULL.

Return value

= NULL Not yet initialized/started.
≠ NULL Pointer to the 8 byte internal ClockIdentity. The DefaultDS ClockIdentity typically never changes, so a pointer to the internal buffer holding the 8 byte ClockIdentity can be used directly for easier code and to avoid needing another buffer.

Additional information

The DefaultDS is initialized during init/start of the PTP service. Until then it returns zero bytes. The own ClockIdentity can also be calculated from the MAC address of the interface as follows (following the standard): MAC[0], MAC[1], MAC[2], 0xFF, 0xFE, MAC[3], MAC[4], MAC[5], MAC[6]

The DefaultsDS ClockIdentity typically is the own identity and can be used to identify a master clock switch/fallback to the own clock as master if no other suitable master is present in the network.

31.9.2 IP_PTP_GetTime()

Description

Retrieves the current PTP time based on the HW (driver) or based on the OS time.

Prototype

```
int IP_PTP_GetTime(IP_PTP_TIMESTAMP * pPTPTimestamp);
```

Parameters

Parameter	Description
pPTPTimestamp	Pointer to an IP_PTP_TIMESTAMP structure (seconds and nanoseconds).

Return value

= 0 Time is synchronized and based on HW time from driver.
= 1 Time is synchronized and based on OS time (no HW driver support).
= 2 PTP module not in slave state. No indication on the validity of the given time.
< 0 Error.

Additional information

All parameter checks have been removed for non-debug builds to ensure a fast access when fetching the timestamp from PTP hardware.

31.9.3 IP_PTP_Halt()

Description

Halts the PTP service based on a configuration context.

Prototype

```
int IP_PTP_Halt(IP_PTP_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Pointer the PTP context. Can be discarded upon succesful halt of the service. Closes previously opened UDP ports if thsi was the last service using them.

Return value

= 0 OK.
< 0 Error, not started.

31.9.4 IP_PTP_Init()

Description

Initializes a PTP context. This should be the first PTP API to call in the application.

Prototype

```
int IP_PTP_Init(      IP_PTP_CONTEXT * pContext,
                const IP_PTP_PROFILE * pProfile);
```

Parameters

Parameter	Description
pContext	Pointer to the PTP context.
pProfile	Pointer to the PTP profile to use. For the moment only "IP_PTP_Profile_1588_2008" is supported.

Return value

= 0 OK.
< 0 Error.

31.9.5 IP_PTP_SetTime()

Description

Sets the PTP time (by applying a positive coarse correction if necessary) and resets the PTP sync state to uncalibrated state. Can be used to set a start time for PTP that is not yet synchronized.

Prototype

```
int IP_PTP_SetTime(IP_PTP_TIMESTAMP * pPTPTimestamp);
```

Parameters

Parameter	Description
pPTPTimestamp	Pointer to an IP_PTP_TIMESTAMP structure (seconds and nanoseconds).

Return value

= 0 O.K.
< 0 Error.

Additional information

A negative coarse correction is not supported. In theory a very large positive coarse correction can be used to make the time wrap around, back to an earlier timestamp. Please consult the manual of your device what about the actual maximum register values for your calculation of the values to apply.

31.9.6 IP_PTP_Start()

Description

Starts the PTP service based on a configuration context.

Prototype

```
int IP_PTP_Start(IP_PTP_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Pointer the PTP context. Needs to stay valid as long as the service is active.

Return value

= 0 OK.
< 0 Error.

31.9.7 IP_PTP_OC_AddMasterFallbackLogic()

Description

Adds a fallback PTP OC logic that is used by drivers that do not come with their own logic in hardware.

Prototype

```
void IP_PTP_OC_AddMasterFallbackLogic(IP_PTP_MASTER * pMaster);
```

Parameters

Parameter	Description
pMaster	Master endpoint that is offered the default OC logic as fallback.

Additional information

Needs to be called before actually adding the master endpoint/role using `IP_PTP_MASTER_Add()`.

31.9.8 IP_PTP_OC_AddSlaveFallbackLogic()

Description

Adds a fallback PTP OC logic that is used by drivers that do not come with their own logic in hardware.

Prototype

```
void IP_PTP_OC_AddSlaveFallbackLogic(IP_PTP_SLAVE * pSlave);
```

Parameters

Parameter	Description
pSlave	Slave endpoint that is offered the default OC logic as fallback.

Additional information

Needs to be called before actually adding the slave endpoint/role using `IP_PTP_SLAVE_Add()`.

31.9.9 IP_PTP_MASTER_Add()

Description

Adds a PTP master to an interface.

Prototype

```
void IP_PTP_MASTER_Add(IP_PTP_CONTEXT * pContext,
                       IP_PTP_MASTER  * pMaster,
                       unsigned         IFaceId);
```

Parameters

Parameter	Description
pContext	Pointer to memory to use for the PTP context.
pMaster	Pointer to management memory of type IP_PTP_MASTER .
IFaceId	Zero-based interface index that shall be used as PTP master.

31.9.10 IP_PTP_MASTER_Config()

Description

Configures parameters for a PTP Master.

Prototype

```
void IP_PTP_MASTER_Config(IP_PTP_MASTER        * pMaster,
                          IP_PTP_MASTER_PARAMS * pParams);
```

Parameters

Parameter	Description
pMaster	Pointer to management memory of type IP_PTP_MASTER .
pParams	Pointer to Master configuration parameters of type IP_PTP_MASTER_PARAMS . The parameters are internally copied.

Additional information

The Master can only be configured before it is added.

31.9.11 IP_PTP_MASTER_Remove()

Description

Removes a PTP master including fallback logic assigned to it and removes it from its assigned `IP_PTP_CONTEXT` as well.

Prototype

```
void IP_PTP_MASTER_Remove(IP_PTP_MASTER * pMaster);
```

Parameters

Parameter	Description
`pMaster`	Pointer to management memory of type `IP_PTP_MASTER` .

31.9.12 IP_PTP_SLAVE_Add()

Description

Adds a PTP slave to an interface.

Prototype

```
void IP_PTP_SLAVE_Add(IP_PTP_CONTEXT * pContext,
                      IP_PTP_SLAVE   * pSlave,
                      unsigned         IFaceId);
```

Parameters

Parameter	Description
pContext	Pointer to memory to use for the PTP context.
pSlave	Pointer to management memory of type IP_PTP_SLAVE .
IFaceId	Zero-based interface index that shall be used as PTP slave.

31.9.13 IP_NI_AddPTPDriver()

Description

Adds an NI specific PTP driver for HW timestamp support.

Prototype

```
int IP_NI_AddPTPDriver(      unsigned        IFaceId,
                       const IP_PTP_DRIVER * pPTPDriver,
                             U32             Clock);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index.
pPTPDriver	PTP driver to add.
Clock	Clock [Hz] of the PTP timer. Can not be 0.

Return value

-1 Error, not supported
0 OK
1 Error, called after driver initialization has been completed.

31.9.14 IP_PTP_OC_SetInfoCallback()

Description

Sets a callback that gets notified on updates of the PTP clock.

Prototype

```
void IP_PTP_OC_SetInfoCallback(IP_PTP_ON_INFO_FUNC * pf);
```

Parameters

Parameter	Description
pf	Callback to be notified about timer corrections.

31.9.15 IP_PTP_OC_SetProductDescription()

Description

Sets the product description of the Ordinary Clock.

Prototype

```
void IP_PTP_OC_SetProductDescription(const char * pDesc);
```

Parameters

Parameter	Description
pDesc	NULL terminated string containing the description. Memory has to stay valid after the call. Maximum 64 characters without NULL termination.

Additional information

Product description contains 3 items separated by a semi-colon:

```
< Manufacturer_name>;<model_number>;<instance_identifier>
```

May be left undefined.

31.9.16 IP_PTP_OC_SetUserDescription()

Description

Sets the user description of the Ordinary Clock.

Prototype

```
void IP_PTP_OC_SetUserDescription(const char * pDesc);
```

Parameters

Parameter	Description
pDesc	NULL terminated string containing the description. Memory has to stay valid after the call. Maximum 128 characters without NULL termination.

Additional information

User description contains 2 items separated by a semi-colon:

< Name_of_the_device>;<Physical_location_of_the_device>

The <Name_of_the_device> field represents a user-defined name for the define (e.g. Sensor1). The <Physical_location_of_the_device> field is a user-defined location (e.g. Rack-2 Shelf-3). One or both of the field could be left unset. For example, valid definitions are: “Sensor1;Rack-2 Shelf-3”, “Sensor1” or “;Rack-2 Shelf-3”.

May be left undefined.

31.9.17 IP_PTP_OC_SetRevision()

Description

Sets the revision of the Ordinary Clock.

Prototype

```
void IP_PTP_OC_SetRevision(const char * pDesc);
```

Parameters

Parameter	Description
pDesc	NULL terminated string containing the description. Memory has to stay valid after the call. Maximum 32 characters without NULL termination.

Additional information

Revision contains 3 items separated by a semi-colon: <HW>;<FW>;<SW>

They correspond to the revision indications of the 3 items. One or more item can be left empty, for example: HW;;SW.

May be left undefined.

31.9.18 IP_PTP_OC_Halt()

Description

Halts the PTP messages processing on the previously started interface. Closes the UDP ports previously opened.

Prototype

```
int IP_PTP_OC_Halt(void);
```

Return value

= 0	OK.
< 0	Error, not started.

31.9.19 IP_PTP_OC_Start()

Prototype

```
int IP_PTP_OC_Start(unsigned IFaceId);
```


31.10 Data structures

31.10.1 IP_PTP_TIMESTAMP

Description

A structure which stores the PTP timestamp. This is the time passed since the EPOCH of the system.

Type definition

```
typedef struct {
  U32  Seconds;
  U32  Nanoseconds;
} IP_PTP_TIMESTAMP;
```

Structure members

Member	Description
Seconds	Time [s] passed since start of epoch of the network, typically since January 1st 1970.
Nanoseconds	The sub-second part of the time passed since the EPOCH expressed in nanoseconds.

31.10.2 IP_PTP_INFO

Description

Returns information about the current PTP status. The Type field has to be evaluated for further information about what part of the union is the information to look at.

Type definition

```
typedef struct {
  IP_PTP_CONTEXT * pContext;
  U8               Type;
} IP_PTP_INFO;
```

Structure members

Member	Description
pContext	PTP context that this information originates from.
Type	Type of information and what part of the union to look at. The Type member is followed by a union that might not be correctly displayed or completely missing in the manual. The following information describes this union part: • IP_PTP_INFO_TYPE_CORRECTION : Information about the latest local time correction can be found in pInfo->Correction in form of IP_PTP_CORRECTION_INFO . • IP_PTP_INFO_TYPE_OFFSET : Information about the offset between slave and master can be found in pInfo->Offset in form of IP_PTP_OFFSET_INFO . • IP_PTP_INFO_TYPE_MASTER_CHANGED: Information about a newly selected master can be found in pInfo->Master or pInfo->MasterNewOld.New in form of IP_PTP_MASTER_INFO . Information about the previous selected master can be found in pInfo->MasterNewOld.Old in form of IP_PTP_MASTER_INFO . • IP_PTP_INFO_TYPE_MASTER_UPDATED: Information about parameter updates for the currently selected master can be found in pInfo->Master or pInfo->MasterNewOld.New in form of IP_PTP_MASTER_INFO . • IP_PTP_INFO_TYPE_MASTER_RESET : Reset to or initial start being our own master. Information can be retrieved in the same way as for IP_PTP_INFO_TYPE_MASTER_CHANGED . pInfo->Master or pInfo->MasterNewOld.New typically contains our own DefaultDS ClockIdentity. pInfo->MasterNewOld.Old contains either the ClockIdentity of the previously selected master (the old master and no other suitable is available, so we fall back to being our own master) or is all zero bytes if the first setup is applied during the init/start phase.

31.10.3 IP_PTP_CORRECTION_INFO

Description

Returns information about the latest local time correction.

Type definition

```
typedef struct {
  U32  Seconds;
  U32  Nanoseconds;
  U32  Flags;
  U8   State;
} IP_PTP_CORRECTION_INFO;
```

Structure members

Member	Description
Seconds	Seconds
Nanoseconds	Nanoseconds
Flags	ORR-ed combination of IP_PTP_FLAGS_*
State	State before applying the correction.

31.10.4 IP_PTP_OFFSET_INFO

Description

Returns information about the offset between slave and master.

Type definition

```
typedef struct {
  U32  Seconds;
  U32  Nanoseconds;
  U32  Flags;
} IP_PTP_OFFSET_INFO;
```

Structure members

Member	Description
Seconds	Seconds
Nanoseconds	Nanoseconds
Flags	ORR-ed combination of IP_PTP_FLAGS_*

31.10.5 IP_PTP_PROT_TYPE

Description

The PTP protocol type can be used to override the default of the system which is typically IPv4/UDP .

Type definition

```
typedef enum {
  IP_PTP_PROT_TYPE_DEFAULT,
  IP_PTP_PROT_TYPE_UDP_IPV4,
  IP_PTP_PROT_TYPE_UDP_IPV6,
  IP_PTP_PROT_TYPE_ETHERNET
} IP_PTP_PROT_TYPE;
```

Enumeration constants

Constant	Description
IP_PTP_PROT_TYPE_DEFAULT	Use the system default which is typically IP_PTP_PROT_TYPE_UDP_IPV4 .
IP_PTP_PROT_TYPE_UDP_IPV4	Use PTP over Layer 3 IPv4/UDP protocol.
IP_PTP_PROT_TYPE_UDP_IPV6	Use PTP over Layer 3 IPv6/UDP protocol.
IP_PTP_PROT_TYPE_ETHERNET	Use PTP over Layer 2 Ethernet protocol.

Additional information

The protocol type is typically set automatically based on the last PTP message received. For a PTP Slave this means that upon receiving a SYNC for example, the DELAY-REQ that is then sent by the Slave will use the same type.

For a Master the protocol type can be set as there will be no previous message being received when sending its SYNC or ANNOUNCE message.

31.10.6 IP_PTP_MASTER_PARAMS

Description

Configuration parameters for a PTP Master.

Type definition

```
typedef struct {
  IP_PTP_PROT_TYPE  Prot;
  I8                LogAnnounceInterval;
  I8                LogSyncInterval;
  I8                LogMinDelayReqInterval;
} IP_PTP_MASTER_PARAMS;
```

Structure members

Member	Description
Prot	Primary protocol to use when sending ANNOUNCE/SYNC messages. Default: IP_PTP_PROT_TYPE_UDP_IPV4 .
LogAnnounceInterval	Log2 interval in which ANNOUNCE messages are sent. For the moment the smaller value of the parameters LogAnnounceInterval and LogSyncInterval is used for both, ANNOUNCE and SYNC messages. Default: 0 (1 second).
LogSyncInterval	Log2 interval in which SYNC messages are sent. For the moment the smaller value of the parameters LogAnnounceInterval and LogSyncInterval is used for both, ANNOUNCE and SYNC messages. Default: 0 (1 second).
LogMinDelayReqInterval	Log2 interval to report to the SLAVE in a DELAY-RESP message when to check the DELAY between Slave and Master. Default: 0 (1 second).

Additional information

Logarithmic time parameters are given as exponent of a two's base of a second (log2) with a range between -128 and +127. Examples:

- LogSyncInterval = 0: Send a SYNC every second.
- LogSyncInterval = 1: Send a SYNC every two seconds.
- LogSyncInterval = -2: Send a SYNC every 250 milliseconds.

31.10.7 IP_PTP_MASTER_INFO

Description

Returns information about a newly selected master or updated parameters with the currently selected one.

Type definition

```
typedef struct {
  U8   abGrandmasterIdentity[];
  U32  UtcOffset;
  U8   IsUtcOffsetValid;
} IP_PTP_MASTER_INFO;
```

Structure members

Member	Description
abGrandmasterIdentity	Master clock ID.
UtcOffset	Offset between TAI and UTC in seconds.
IsUtcOffsetValid	Is the offset valid ? Basically this means, has this offset been updated/set by a master ?

31.11 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the PTP client presented in the tables below have been measured on a Cortex-M4 system with the default configuration.

31.11.1 ROM usage on a Cortex-M4 system

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

Addon	ROM
emNet PTP client	approximately 7.0 kBytes

31.11.2 RAM usage

Addon	RAM
emNet PTP client	approximately 300 Bytes

Chapter 32

NTP client (Add-on)

The emNet implementation of the Network Time Protocol (NTP) client is an optional extension to emNet. It can be used to get current time from NTP servers. All functions that are required to add NTP client functionality to your application are described in this chapter.

32.1 emNet NTP client

The emNet NTP client implementation is an optional extension which can be seamlessly integrated into your application. The NTP client implementation allows an embedded system to use real timestamps from a remote NTP server without using a without using a Real Time Clock (RTC). The NTP protocol is based on NTP v4.

The NTP client module implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 5905]	Network Time Protocol Version 4: Protocol and Algorithms Specification Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc5905.txt*

The following table shows the contents of the emNet NTP client root directory:

Directory	Content
`.\IP\`	Contains the NTP client sources `IP_NTP_CLIENT.c`.

32.2 Feature list

- Low memory footprint.
- Seamless integration with the emNet stack.
- Time synchronization with remote NTP servers.

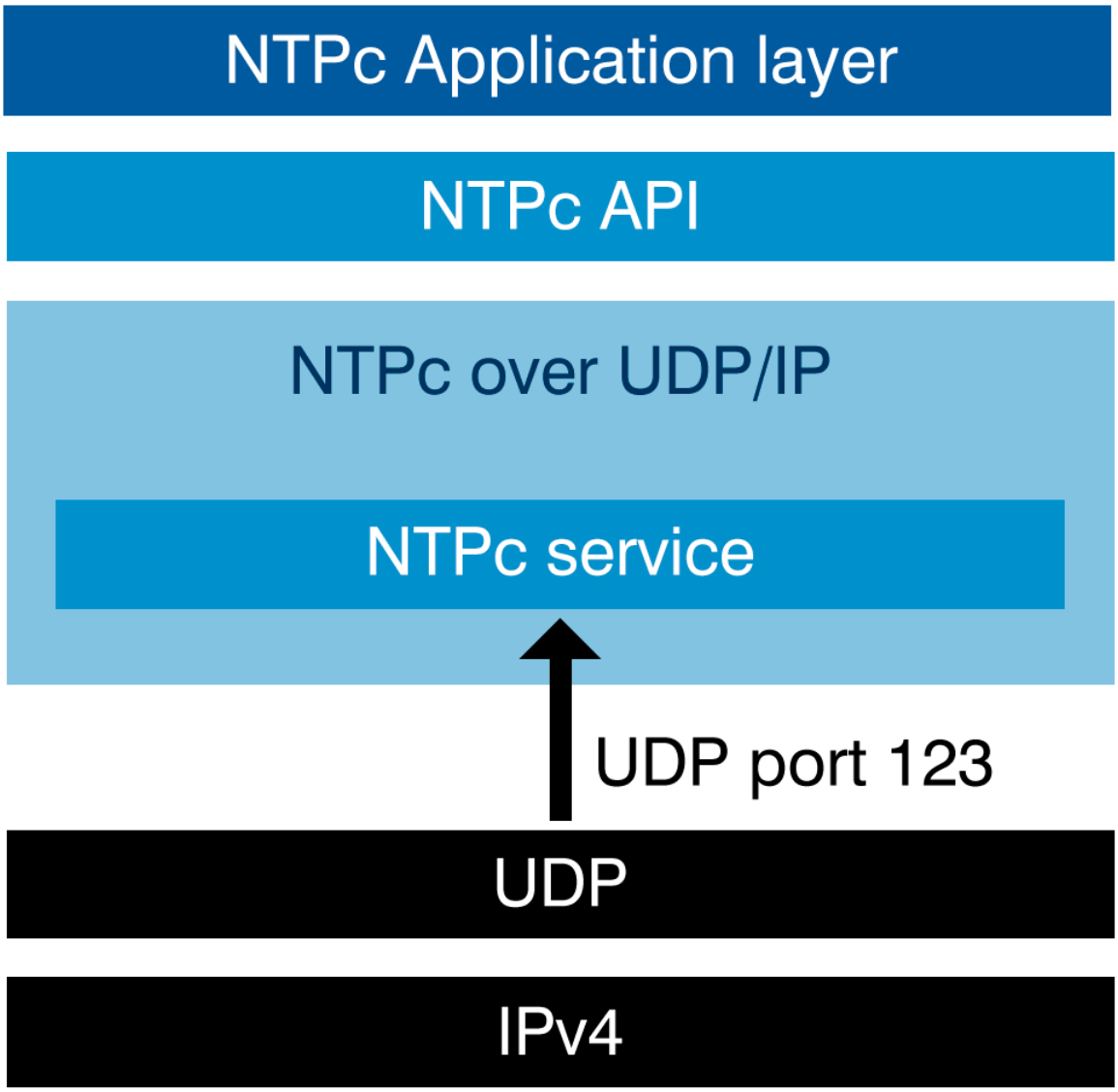
32.3 Requirements

TCP/IP stack

The emNet NTP client implementation requires the emNet TCP/IP stack. In order to use a server pool, a DNS environment needs to be present and configured.

32.4 NTP backgrounds

The NTP protocol is a service on top of the UDP/IP protocol, it provides a way of synchronizing the target time with a local or remote NTP server over the network.



Using NTP, an emNet 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.

32.4.1 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.

Instead of using this structure, a unsigned 64 bits variable (U64) bit could be used with seconds in the upper 32 bits part and fraction in the lower 32 bits part.

More information about the NTP timestamp can be found in *RFC 5905*.

Warning

> In order to work, the emNet NTP client add-on requires timestamp in IP packets. The switch `IP_SUPPORT_PACKET_TIMESTAMP` shall be set to 1.

32.4.2 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 the offset to the OS clock is initialized at January 2017.

32.4.3 Algorithm and memory

Although the time conversion is quite simple, the NTP algorithm requires many computation and memory usage. From a measurement on a server, some parameters are computed (like delay, dispersion,…) and for every monitored server the last height of these parameters should be kept to enter the mitigation algorithm which will choose the best clocks to use for the time correction.

The full RFC implementation cost many RAM and computation power. In order to mitigate this issue on embedded targets, emNet NTP client add-on proposes along with the full RFC implementation, a simpler (not RFC compliant) but less demanding implementation, activated through a compilation switch.

Another bias to the standard is that the NTP module will always be a client only. It always indicates itself as not synchronized to the other clocks in the network, so it won't be used as a reference.

32.4.4 NTP server pool

To synchronize its time, the emNet NTP client uses server clocks. There are three sources to populate its list of server clocks that work simultaneously:

- User defined clock through the APIs `IP_NTP_CLIENT_AddServerClock()` and `IP_NTP_CLIENT_AddServerClockIPv6()`.
- Local clocks which broadcast periodically their time on UDP port 123 will also be used.
- Usage of server pool.

A server pool is a network name (i.e. "0.pool.ntp.org"). When a DNS request is sent for its address, the DNS reply provides in general four IP addresses of four different clocks.

In order to use this pool mechanism request, a DNS server should be present in the network and its address known by emNet either by configuration or given by the DHCP server for example.

32.4.5 Time function

The emNet NTP client add-on uses an IP callback to retrieve the current time in microseconds. By default a function is provided which uses `IP_OS_GetTime32()` in milliseconds. It is possible to provide a clock with a better precision using `IP_SetMicrosecondsCallback()`. For example when using embOS:

```
IP_SetMicrosecondsCallback(OS_GetTime_us64);
```


32.5 NTP client configuration

The emNet NTP 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.

32.5.1 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).

32.5.2 Configuration switches

Type	Symbolic name	Default	Description
B	`NTP_USE_SIMPLER_VERSION`	1	As the RFC implementation is quite demanding in resources, activating this switch makes the NTP add-on use less memory and computation, but it is not strictly speaking RFC compliant as the algorithm used to select reference clocks is simpler.
B	`IP_SUPPORT_PACKET_TIMESTAMP`	0	NTP needs timestamping capabilities in `IP_PACKET`. To use emNet NTP add-on, this switch shall be set to 1.
N	`NTP_MAX_POOL`	4	Maximum number of pool to be configured.
N	`NTP_MAX_SERVER`	8	Maximum number of server clock to monitor.
N	`NTP_POOL_RESOLUTION_TIMEOUT`	2000	Timeout in ms to wait for the DNS reply after having sent a request to a server pool.
N	`NTP_SERVER_REACH`	4	Number of attempts to connect to a server clock before giving up.

32.6 API functions

Function	Description
NTP client	
IP_NTP_CLIENT_Start()	Starts the NTP processing and open the corresponding UDP ports.
IP_NTP_CLIENT_Halt()	Stops the NTP message processing and closes the corresponding UDP ports.
IP_NTP_CLIENT_ResetAll()	Resets all internal contexts.
IP_NTP_CLIENT_Run()	NTP management function to be called periodically.
IP_NTP_CLIENT_AddServerPool()	Adds a server pool where to look for clocks.
IP_NTP_CLIENT_FavorLocalClock()	Sets the flag to favor clock found on local network and user defined clocks.
IP_NTP_CLIENT_AddServerClock()	Adds a user defined clock by its IPv4 IP address.
IP_NTP_CLIENT_AddServerClockIPv6()	Adds a user defined clock by its IPv6 IP address.
IP_NTP_GetTimestamp()	Returns the current NTP time in timestamp format.
IP_NTP_GetTime()	Returns the current NTP time in NTP U64 format.

32.6.1 IP_NTP_CLIENT_Start()

Description

Starts the NTP processing and open the corresponding UDP ports.

Prototype

```
int IP_NTP_CLIENT_Start(void);
```

Return value

= 0	Success.
< 0	Error.

Additional information

In order for NTP to work, it is needed to call first `IP_NTP_CLIENT_Start()` and then periodically call `IP_NTP_CLIENT_Run()` .

Example

Example of a code that starts the emNet NTP client, runs it and gets every 5s the current time:

```
Time = IP_OS_GetTime32() + 5000;
//
// Start the NTP client.
//
IP_NTP_CLIENT_Start();
//
for (;;) {
  //
  // Run the NTP state machine.
  //
  IP_NTP_CLIENT_Run();
  //
  // Every 5s, print the current time.
  //
  if (IP_IsExpired(Time) != 0) {
    //
    // Next expiry in 5s.
    //
    Time += 5000;
    //
    // Get the NTP time.
    //
    NTPTime = IP_NTP_GetTime(&Status);
    //
    // Pretty print of the current time.
    //
    _PrintNTPTime(NTPTime, Status);
  }
  //
  // Give some time to other lower priority tasks to run.
  //
  OS_Delay(10);
}
```


32.6.2 IP_NTP_CLIENT_Halt()

Description

Stops the NTP message processing and closes the corresponding UDP ports.

Prototype

```
int IP_NTP_CLIENT_Halt(unsigned ClearUserDefined);
```

Parameters

Parameter	Description
ClearUserDefined	Flag to clear or keep user defined clocks.

Return value

= 0 Success.
< 0 Error.

Additional information

Halt doesn't reset the time offset thus a call to `IP_NTP_GetTime()` or `IP_NTP_GetTime-stamp()` is still possible.

32.6.3 IP_NTP_CLIENT_ResetAll()

Description

Resets all internal contexts. This also resets the variables that are otherwise untouched by a simple halt.

Prototype

```
int IP_NTP_CLIENT_ResetAll(void);
```

Return value

= 0	Success.
< 0	Error, NTP is currently running.

32.6.4 IP_NTP_CLIENT_Run()

Description

NTP management function to be called periodically.

Prototype

```
int IP_NTP_CLIENT_Run(void);
```

Return value

= 0	Success.
< 0	NTP is not initialized (no call to start).

See *IP_NTP_CLIENT_Start* on page 880 for an example.

32.6.5 IP_NTP_CLIENT_AddServerPool()

Description

Adds a server pool where to look for clocks.

Prototype

```
int IP_NTP_CLIENT_AddServerPool(      unsigned   IFaceId,
                                const char     * sPool);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index on which the pool could be reached.
sPool	Null terminated string with the pool address.

Return value

= 0 The pool was added.
< 0 The pool was not added (list is full)

Additional information

It is possible to add NTP_MAX_POOL pool.

The string gives the address of the pool, for example “0.pool.ntp.org”.

In order to resolve the pool and get servers addresses, a DNS request is sent. It is needed to have a DNS server in the network to use this feature.

Example

```
IP_NTP_CLIENT_Start();
//
// Configure maximum 4 pools.
//
IP_NTP_CLIENT_AddServerPool(_IFaceId, "0.pool.ntp.org");
IP_NTP_CLIENT_AddServerPool(_IFaceId, "1.pool.ntp.org");
IP_NTP_CLIENT_AddServerPool(_IFaceId, "2.pool.ntp.org");
IP_NTP_CLIENT_AddServerPool(_IFaceId, "3.pool.ntp.org");
```


32.6.6 IP_NTP_CLIENT_FavorLocalClock()

Description

Sets the flag to favor clock found on local network and user defined clocks.

Prototype

```
void IP_NTP_CLIENT_FavorLocalClock(unsigned OnOff);
```

Parameters

Parameter	Description
OnOff	Flag to set.

Additional information

Clock on local network should broadcast regularly their time to be identified.

When the full RFC is used, this has only an impact on the survivors list. A local clock would be anyway discarded if it doesn’t pass the clock filter algorithm.

32.6.7 IP_NTP_CLIENT_AddServerClock()

Description

Adds a user defined clock by its IPv4 IP address.

Prototype

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

Parameters

Parameter	Description
IFaceId	Zero-based interface index on which the clock is reachable.
IPAddr	IP address in host byte order.

Example

```
//
// Configure a local clock.
//
IP_NTP_CLIENT_AddServerClock(_IFaceId, IP_BYTES2ADDR(192, 168, 5, 10));
```


32.6.8 IP_NTP_CLIENT_AddServerClockIPv6()

Description

Adds a user defined clock by its IPv6 IP address.

Prototype

```
void IP_NTP_CLIENT_AddServerClockIPv6(unsigned   IFaceId,
                                      U8       * pIPAddr);
```

Parameters

Parameter	Description
IFaceId	Zero-based interface index on which the clock is reachable.
pIPAddr	Pointer to the 16 bytes of the IPv6 address.

Notes

To use this function, the IPv6 add-on should be present and activated (IP_SUPPORT_IPV6 is 1).

32.6.9 IP_NTP_GetTimestamp()

Description

Returns the current NTP time in timestamp format.

Prototype

```
int IP_NTP_GetTimestamp(IP_NTP_TIMESTAMP * pTimestamp);
```

Parameters

Parameter	Description
pTimestamp	Pointer to the structure filled with the current time.

Return value

-1 NTP is not started.
1 NTP is started but not synced to a master clock.
0 NTP is started and synced.

Refer to *Structure IP_NTP_TIMESTAMP* on page 832 in SNTP chapter.

32.6.10 IP_NTP_GetTime()

Description

Returns the current NTP time in NTP U64 format.

Prototype

```
U64 IP_NTP_GetTime(int * pStatus);
```

Parameters

Parameter	Description
pStatus	Variable giving indication on the time validity (could be NULL). • -1: NTP is not started. • 1: NTP is started but not synced to a master clock. • 0: NTP is started and synced.

Return value

Current time in NTP 64 bits format.

Additional information

The current time in NTP 64 bits format contains the seconds in the higher 32 bits and the fraction part in the lower 32 bits.

32.7 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the NTP client presented in the tables below have been measured on a Cortex-M4 system with the default configuration.

32.7.1 Full RFC configuration

The switch `NTP_USE_SIMPLER_VERSION` is set to 0.

32.7.1.1 ROM usage on a Cortex-M4 system full RFC

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

Addon	ROM
emNet NTP client with IPv6	approximately 4.8 kBytes
emNet NTP client without IPv6	approximately 4.5 kBytes

32.7.1.2 RAM usage full RFC

Addon	RAM
emNet NTP client with IPv6	approximately 3.4 kBytes
emNet NTP client without IPv6	approximately 3.2 kBytes

32.7.2 Simpler configuration

The switch `NTP_USE_SIMPLER_VERSION` is set to 1 (default).

32.7.2.1 ROM usage on a Cortex-M4 system simpler version

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

Addon	ROM
emNet NTP client with IPv6	approximately 3.4 kBytes
emNet NTP client without IPv6	approximately 3.1 kBytes

32.7.2.2 RAM usage simpler version

Addon	RAM
emNet NTP client with IPv6	approximately 0.8 kBytes
emNet NTP client without IPv6	approximately 0.6 kBytes

Chapter 33

SNMP Agent (Add-on)

The emNet Simple Network Management Protocol (SNMP) Agent is an optional extension to emNet. The SNMP Agent can be used with emNet 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 emNet 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.

33.1 emNet SNMP Agent

The emNet 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 3411]	An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc3411.txt*
[RFC 3412]	Message Processing and Dispatching for the Simple Network Management Protocol (SNMP) Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc3412.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*
[RFC 5343]	“Simple Network Management Protocol (SNMP) Context EngineID Discovery” Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc5343.txt*

The following table shows the contents of the emNet SNMP Agent root directory:

Directory	Content
`.\Application\`	Contains the example application to run the SNMP Agent with emNet.
`.\Config\`	Contains the SNMP Agent configuration file. Refer to *SNMP Agent configuration* on page 914 for detailed information.
`.\CRYPTO\`	Cryptographic hash and/or encrypt/decrypt modules required for SNMPv3 support. Only included with the SNMPv3 extension.
`.\Doc\`	Contains release notes and documentation.
`.\IP\`	Contains the SNMP Agent sources and header files, `IP_SNMP_AGENT*`.
`.\SEGGER\`	Contains SEGGER helper routines.
`.\Shared\`	Modules shared between multiple SEGGER products or application samples.
`.\Windows\IP\SNMP_Agent\`	Contains the source, the project files and an executable to run the emNet SNMP Agent on a Microsoft Windows host. Refer to *Using the SNMP Agent samples* on page 907 for detailed information.

33.2 Feature list

- Low memory footprint.
- Only few SNMP knowledge required.
- Easy MIB tree setup.
- Supports SNMPv1 and SNMPv2c.
- SNMPv3 available as optional component.
- Supports SNMPv1 & SNMPv2 TRAP messages.
- Supports SNMPv2 INFORM messages.
- SNMPv3 RFC 3414 authentication support.
- SNMPv3 RFC 3414 privacy support.
- SNMPv3 Engine discovery supported.
- MIB-II support (System and Interfaces branch) for emNet 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.

33.3 SNMP Agent requirements

TCP/IP stack

The emNet SNMP Agent requires an UDP/IP stack. It is optimized for emNet, 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 emNet as well as an implementation which uses the zero-copy API of emNet.

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

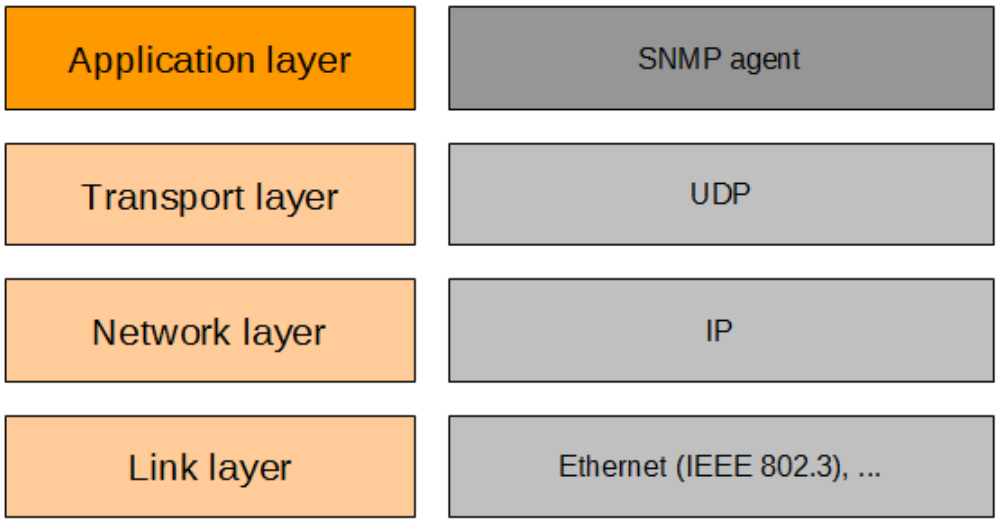
The PEN used in the samples (dec. 46410) is the PEN registered for SEGGER Microcontroller GmbH & Co. KG. This needs to be be changed to your own PEN in your product and the content of the MIB is subject to change in the future.

Some API might require the PEN in byte BER encoding. An easy way to encode your decimal PEN to byte BER is to use the `IP_SNMP_AGENT_EncodeOIDValue()` API.

33.4 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 emNet SNMP Agent is v1 and v2c. The latest protocol version is v3.

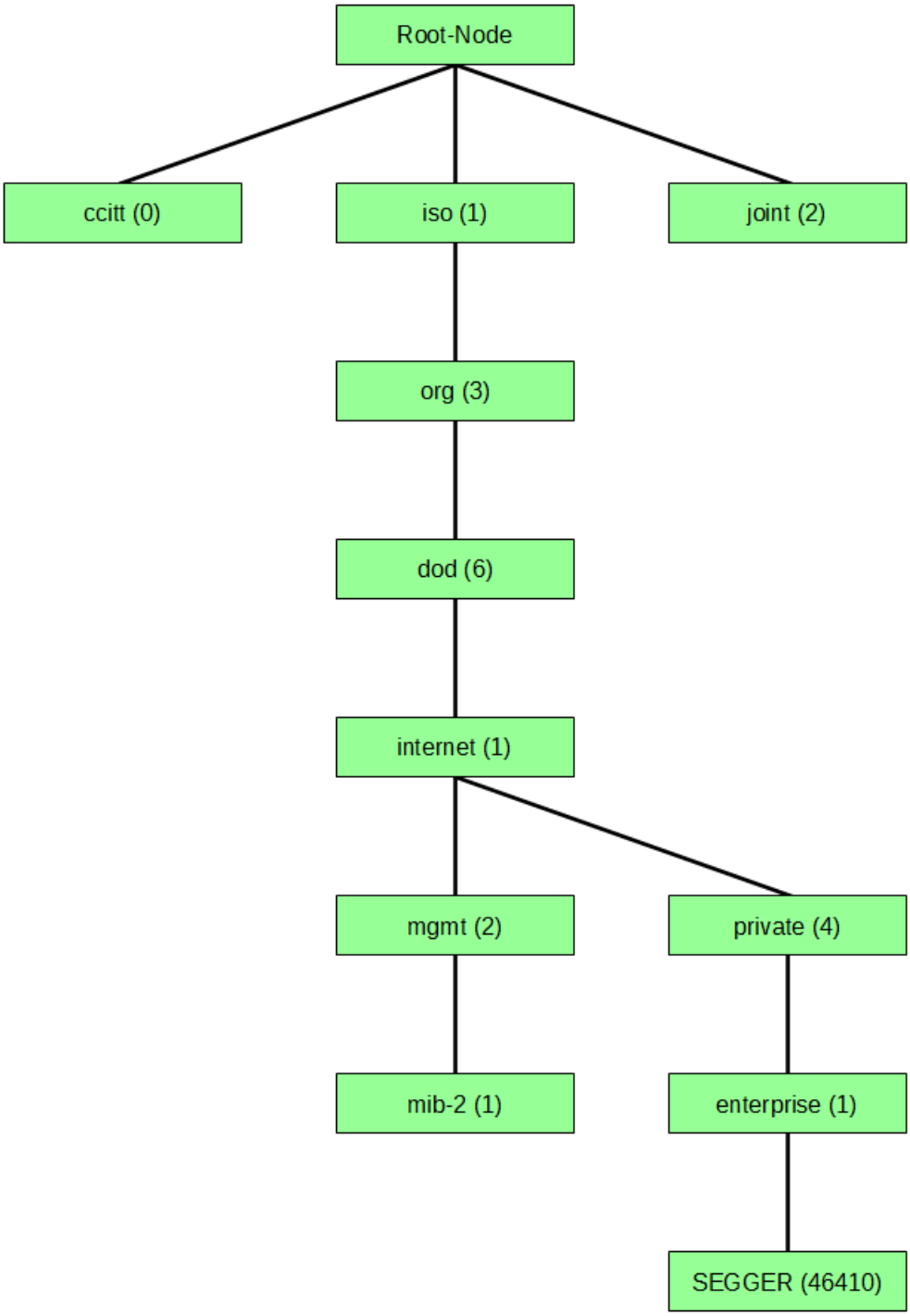
The SNMP Agent is the part that is implemented directly inside a device to fulfill requests like status information and configuration of the device.



33.4.1 Data organization in SNMP

SNMP data is organized in Management Information Base (MIB) blocks. The blocks itself are typically called MIB 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.

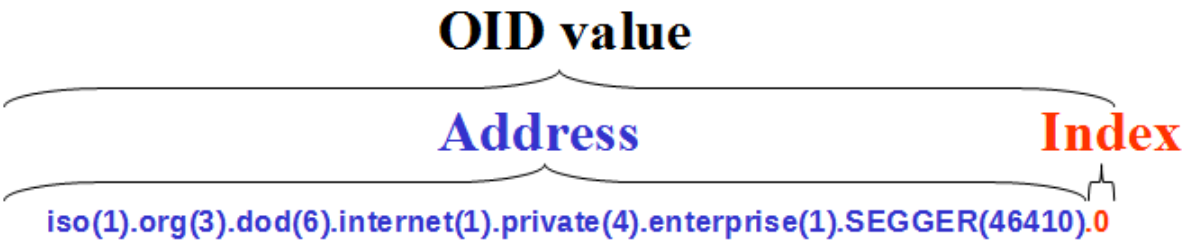
A typical MIB tree for access to your own enterprise specific data is shown below. In this example the MIBs from the root node down to the SEGGER enterprise MIB are shown.



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.

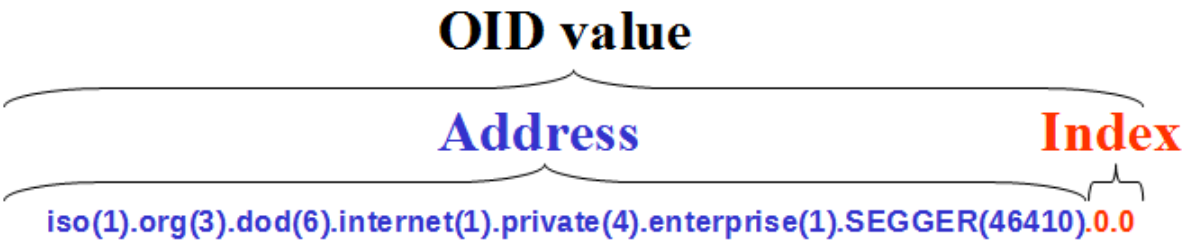
33.4.2 OID value, address and index

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.

33.4.3 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.

33.4.3.1 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.

33.4.3.2 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.

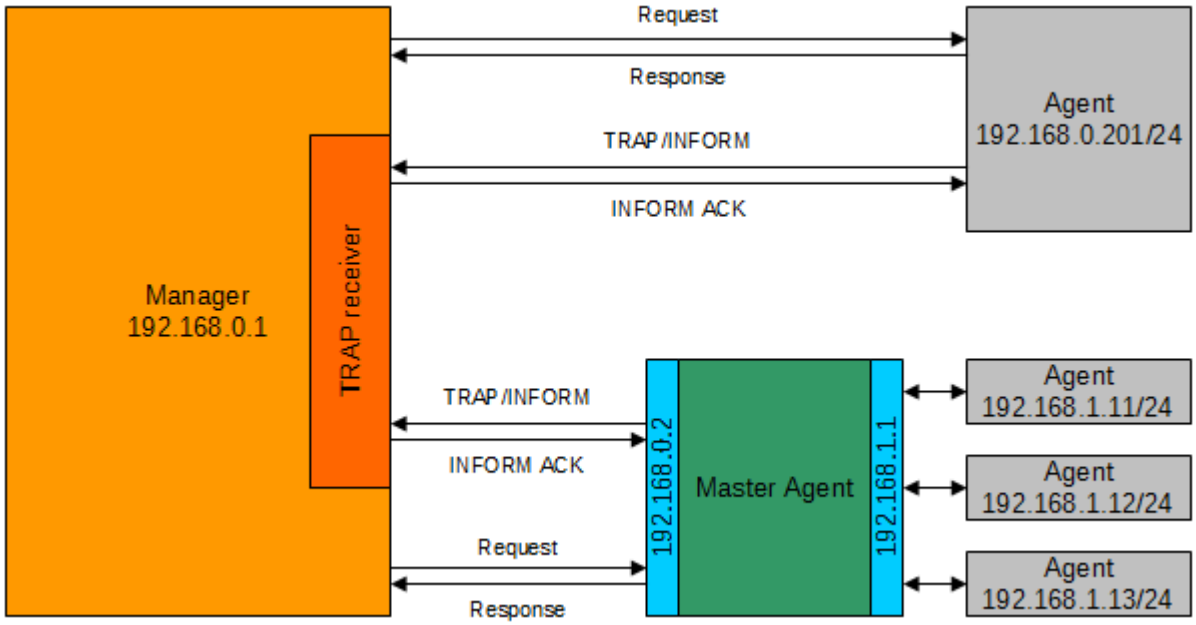
Data Type	ID	Description
Integer32	0x02	Signed 32-bit integer. SNMPv2 version of the INTEGER data type.
BITS	0x04	Array of bit values sent in an U8 data array. Basically the same as the OCTET STRING data type.
Counter32	0x41	Unsigned 32-bit value, non decreasing. SNMPv2 version of Counter. Only supported 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.

33.4.4 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 for a response sent back. The only exception to this behavior is a TRAP 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 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:



33.4.5 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
`get-request`	Requests one or more values from specific OID values.
`getnext-request`	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.
`get-response`	Responses sent back upon any request message received.
`set-request`	Sets one or more values for specific OID values.
`TRAPv1`	Unrequested message sent from Agent to one or more Managers. Does not check for reception of the message.

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
`getbulk-request`	Used for sending one or multiple getnext-requests to retrieve a large amount of data.
`TRAPv2`	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.
`INFORM`	Unrequested message sent from Agent to one or more Managers. Awaits an acknowledge sent back from the Manager and is re-transmitted if the acknowledge is not received.

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 stand 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.

SNMPv3

SNMPv3 adds support for extensible security layers while in general keeping the same message types as SNMPv2c. It supports the following new message type:

Type	Description
`REPORT`	SNMP messages can let the receiver know that they would like to receive a report in case the message was not accepted due to some parameters the sender has used, maybe due to not knowing them when originally formulating the first iteration of the message. This includes timestamps for replax protection of messages as well as knowing more details about which SNMP instance to address as receiver exactly.

While SNMPv2c introduced a simple security mechanism it is subject to being transmitted in cleartext and at the same time only serves a very simple single password protection without any hardening against replay attacks. SNMPv3 allows different combinations of security layers while providing simple enough protection against replay attacks to not over complicate things while still providing a small enough time window to prevent most replay attacks that are typically executed based on recorded messages long after the original message has been sent originally.

The most common and widely used security concept of the User-based Security Model (USM) provides username based access with optional support for a password/key based authentication of messages as well as optional privacy/encryption on top of the authentication, with the authentication being done last to authenticate the whole encrypted message.

33.4.6 SNMPv3 specific information

While SNMPv3 messages in their principles operate in the same way as with previous SNMP protocol versions, SNMPv3 and the USM security layer introduce the concept of an authoritative SNMP Engine. An SNMP Engine acts as a handler to receive and create SNMP messages and was introduced with SNMPv3 to introduce a layered message structure that can be extended in the future.

Differences between SNMPv3 and older version message types

SNMPv3 in general uses the same message/PDU types as in older SNMP versions. The headers before the PDU content (the part with the VarBinds of OIDs and values) is however no longer of a fixed structure rather than being constructed out of several security layers defined by the security model selected for the Engine.

The User-based Security Model (USM)

The User-based Security Model (USM) used with SNMPv3 is the standard security model used with SNMPv3. As the name implies it uses a user database that is managed by a so called "SNMP Engine". Different permissions and security levels in terms of AUTH(entication) and PRIV(acy) aka encryption can be assigned to each user.

SNMPv3 (USM) Engine identifier

The Engine used by the User-based Security Model (USM) for authoritative purposes in a message is defined by RFC 3411 and RFC 3412 . The EngineId used in a message is typically the EngineId of the receiver of the message which typically is the Agent receiving a request.

The "SnmpEngineId" OCTET STRING in RFC 3412 is not limited in length and different implementations exist. RFC 3411 specifies the "SnmpEngineID" as "OCTET STRING (SIZE(5..32))". While we are not limited in the length of foreign "SnmpEngineID" fields in SNMP messages due to how we parse the message, it is suggested to stick to an RFC 3411 conform "SnmpEngineID". An EngineId based on a 6 byte MAC address is constructed as follows:

- 4 bytes consisting of the PrivateEnterpriseNumber (PEN) with bit 31 set to indicate the EngineId is following RFC 3411 .
- 1 byte with value `0x03`, indicating that the rest of the EngineId is a 6 byte MAC address.
- 6 bytes MAC address.

Example for the EngineId for the SEGGER PEN 46410:
`0x80 0x00 0xB5 0x4A 0x03` <6 byte MAC address>

Engine discovery and REPORT messages

The (primary) Engine used by an SNMP participant as well as its EngineId and further details can be discovered by sending an empty get-request messages that has the REPORT flag set. If the receiver supports SNMPv3 messages it responds back with a REPORT messages containing its EngineId and depending on other factors like steps done towards a successful authentication with the peer more information to utilize deeper security layers.

If only the IP address of an SNMPv3 peer is known its Engine has to be first discovered before being able to send meaningful SNMPv3 requests to it. Depending on the security level requested by the peer Engine it might also require us to utilize its current time that it will tell us in a REPORT when sending a message. Sending a message outside of the authentitative Engines time window generates a REPORT with the new time to use to prevent replay attacks with messages that are considered as too old to be trusted.

TRAP/INFORM usage of the EngineId

While the EngineId is typically the Engine of the message receiver, this is not true for TRAP messages to stay true to their original meaning of an unconfirmed notification. For this purpose the authentitative Engine used in a TRAP message is the Engine of the sender. The receiver needs to be able to authenticate and maybe even decrypt the message if required by exactly knowing the details about the sender Engine, the username used within the

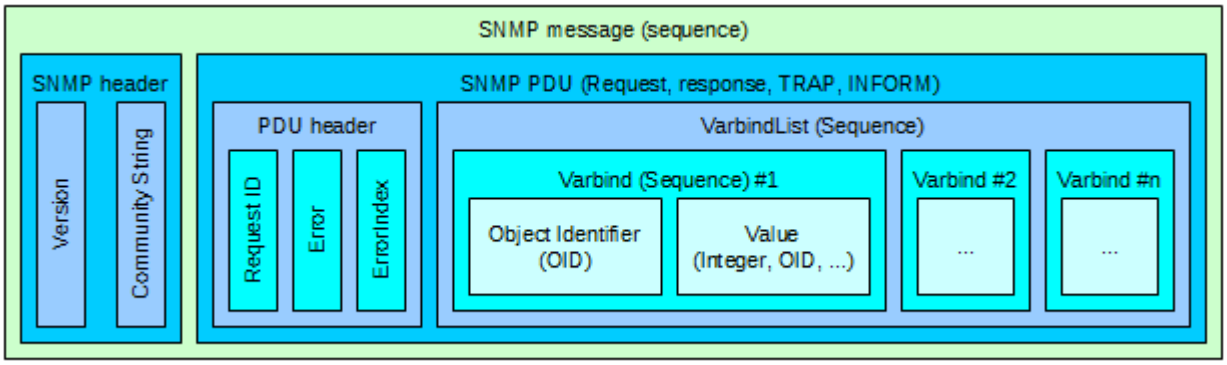
message and the potential authentication and privacy settings used by this user. Except for receiving an encrypted message a receiver might simply trust the content of a TRAP received as the authentitative Engine is of the sender anyhow.

As INFORM messages technically behave like a set-request followed by a get-response from the receiver, the authentitative EngineId used in INFORM messages is once again the Engine of the receiver as it is used for any regular request.

33.4.7 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 defines list of one or more Managers.

The encoding used in SNMP messages is a binary encoding, so the data can not be read clear-type. An SNMP message consists of several nested elements as can be seen in the following example picture:



33.4.8 SNMP Agent return codes

The emNet SNMP Agent API distinguishes two types of return codes.

- Generic SNMP protocol error codes
- emNet SNMP Agent API error codes

Both types might be returned by the emNet 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 emNet SNMP Agent API.

Return code	Value	Description
IP_SNMP_OK	0	Everything O.K.
IP_SNMP_ERR_TOO_BIG	1	Either the request received was too big to parse or the response to send does not fit into one SNMP message.
IP_SNMP_ERR_NO_SUCH_NAME	2	The OID value to access does not exist.
IP_SNMP_ERR_BAD_VALUE	3	Syntax or value error.
IP_SNMP_ERR_GENERIC	5	Generic error, not further specified.
IP_SNMP_ERR_NO_ACCESS	6	The OID value to access is not available. Might be due to not being visible for the supplied community string.
IP_SNMP_ERR_WRONG_TYPE	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 conatins an Unsigned32 value to set.
IP_SNMP_ERR_NO_CREATION	11	The OID value to access in a set-request does not exist and the Agent is unable to create it.
IP_SNMP_ERR_AUTH	16	No access to this OID value, e.g. wrong community string.

emNet SNMP Agent API return codes

These return codes are used by the emNet SNMP Agent API to return the result of internal processes such as parsing results and generating responses.

Return code	Value	Description
IP_SNMP_AGENT_OK	0	Everything O.K.
IP_SNMP_AGENT_ERR_MISC	-1	Generic error, not further specified.
IP_SNMP_AGENT_ERR_UNSUPPORTED_VERSION	-2	A message has been received with an unsupported protocol version.
IP_SNMP_AGENT_ERR_AUTH	-3	Wrong community string received in request. Requested action is not allowed.
IP_SNMP_AGENT_ERR_MALFORMED_MESSAGE	-4	Error while parsing a received message.
IP_SNMP_AGENT_ERR_TOO_BIG	-5	Response to send back is too big to send.

33.5 Using the SNMP Agent samples

Ready to use examples for Microsoft Windows and emNet are supplied. If you use another UDP/IP stack, the samples have to be adapted.

The supplied sample application `IP_SNMP_AGENT_Start.c` and `IP_SNMP_AGENT_Start_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 emNet do the same for the SNMP Agent. However they differ in the way they are using the UDP/IP stack used for communication.

33.5.1 IP_SNMP_AGENT_Start.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.

33.5.2 IP_SNMP_AGENT_Start_ZeroCopy.c

This sample uses the emNet 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 emNet for sending without first having to copy the payload into another buffer.

As the packet buffers are pre-allocated in emNet 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.

33.5.3 Using the Windows SNMP Agent 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 emNet 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.

33.5.4 Features of the SNMP Agent 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 *SNMP Agent requirements* on page 894.

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.

33.5.5 SNMPv3 samples

The SNMPv3 features are built into the samples and can be enabled by changing the configuration define `SUPPORT_SNMPV3` in the samples. Dedicated versions of the samples with SNMPv3 enabled might exist as well.

When enabling SNMPv3 support in the applications a variety of SNMPv3 users are added. The users added and their capabilities should be looked up in the application sample directly.

33.5.6 Testing the SNMP Agent 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. The majority of the examples are shown for SNMPv2c as except for some additional parameters they are the same for SNMPv3 with these tools.

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
```

Clear the first LED with SNMPv3

With SNMPv3 various combinations of security are possible. It is not subject of this documentation to list all possible combination but to give an example of how to test the provided examples in principle.

The combination of optional authentication and privacy depends on the specific user of the respective authoritative SNMP Engine. Therefore the following command line is only one example for the provided application example and the SNMPv3 users created in it. For a sample command line for each user created in the example application please refer to the application routine creating the users. For each user created a sample command line is stated.

```
snmpset -v3 -u testuser_authPriv_MD5 -l authPriv -a MD5 -A AUTHpass -x DES -X
 PRIVpass <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 emNet SNMP Agent. You will need to configure the IP address of the host running the TRAP browser in the SNMP Agent sample.

33.6 The MIB callback

To each MIB of the emNet 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 getnext-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_cbSample()`:

```
/*********************************************************************
*
*       _SNMP_cbSample()
*
*  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_cbSample(      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 folowing 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;
}
```


33.7 SNMP Agent configuration

The emNet 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.

33.7.1 SNMP Agent 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).

33.7.2 SNMP Agent compile time configuration switches

Type	Symbolic name	Default	Description
F	`IP_SNMP_AGENT_WARN`	--	Defines a function to output warnings. In debug configurations (`DEBUG = 1`) `IP_SNMP_AGENT_WARN` maps to `IP_Warnf_Application()`.
F	`IP_SNMP_AGENT_LOG`	--	Defines a function to output logging messages. In debug configurations (`DEBUG = 1`) `IP_SNMP_AGENT_LOG` maps to `IP_Logf_Application()`.
F	`IP_SNMP_AGENT_MEMCPY`	memcpy	memcpy function.
B	`IP_SNMP_AGENT_SUPPORT_PANIC_CHECK`	Debug: 1 Release: 0	Defines if upon a critical error the execution shall be halted.
B	`IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES`	1	Defines if 64-bit types like double and relatives like float are supported.
N	`IP_SNMP_AGENT_WORK_BUFFER`	64	Defines the size of the buffer used to do internal work like assembling OID values for further usage.

33.8 API functions

Function	Description
IP_SNMP_AGENT_AddCommunity()	Adds a community string for access rights management.
IP_SNMP_AGENT_AddMIB()	Adds a MIB to the tree.
IP_SNMP_AGENT_AddInformResponse-Hook()	Registers a hook that is called upon a state change of an INFORM item waiting for an ACK.
IP_SNMP_AGENT_CancelInform()	Cancels an INFORM message that might be still in process of resending.
IP_SNMP_AGENT_CheckInformStatus()	Retrieves the status of an INFORM message sent.
IP_SNMP_AGENT_DeInit()	Deinitializes the SNMP Agent.
IP_SNMP_AGENT_Exec()	Executes management tasks.
IP_SNMP_AGENT_GetMessageType()	Retrieves the message type of the message in the given buffer.
IP_SNMP_AGENT_Init()	Initializes the SNMP Agent.
IP_SNMP_AGENT_PrepareTrapInform()	Prepares a TRAP/INFORM message.
IP_SNMP_AGENT_ProcessInformResponse()	Processes a response that has been received for a previously sent INFORM message.
IP_SNMP_AGENT_ProcessMessage()	Processes a received message and fills the output buffer with the response for sending back.
IP_SNMP_AGENT_SendTrapInform()	Sends a TRAP/INFORM message.
IP_SNMP_AGENT_SetCommunityPerm()	Sets permissions for a community profile.
SNMPv3 specific functions	
IP_SNMP_AGENT_MPV3_Add()	Adds the MessageProcessor (MP) for SNMPv3 messages.
IP_SNMP_AGENT_SetInformReportCallback()	Sets a callback that is executed when information about an SNMPv3 engine are received.
IP_SNMP_AGENT_SM_USM_Add()	Adds the User-basedSecurityModel (USM) processor for SNMPv3 messages.
IP_SNMP_AGENT_SM_USM_CalcKey()	Calculates the AuthKey or PrivKey for a password and SNMP EngineId.
IP_SNMP_AGENT_SM_USM_SetUserTable()	Sets the SNMPv3 user table.
Standard MIB tree setup functions	
IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2Interfaces()	Adds the base MIBs iso(1).org(3).dod(6).internet(1).ietf(2).mib2(1).interfaces(2) to the tree.
IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetIetfMib2System()	Adds the base MIBs iso(1).org(3).dod(6).internet(1).ietf(2).mib2(1).system(1) to the tree.
IP_SNMP_AGENT_AddMIB_IsoOrgDodInternetPrivateEnterprise()	Adds the base MIBs iso(1).org(3).dod(6).internet(1).private(4).enterprise(1) to the tree.
Message construct functions	

Function	Description
IP_SNMP_AGENT_CloseVarbind()	Closes a previously opened Varbind after an OID and value pair has been stored and inserts the length into the Varbind header.
IP_SNMP_AGENT_OpenVarbind()	Prepares a Varbind in the output buffer of the given context.
IP_SNMP_AGENT_StoreBits()	Stores a bitfield into an SNMP message.
IP_SNMP_AGENT_StoreCounter()	Stores a Counter into an SNMP message.
IP_SNMP_AGENT_StoreCounter32()	Stores a Counter32 into an SNMP message.
IP_SNMP_AGENT_StoreCounter64()	Stores a Counter64 into an SNMP message.
IP_SNMP_AGENT_StoreCurrentMibOidAndIndex()	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.
IP_SNMP_AGENT_StoreDouble()	Stores a double-precision float into an SNMP message.
IP_SNMP_AGENT_StoreFloat()	Stores a single-precision float into an SNMP message.
IP_SNMP_AGENT_StoreGauge()	Stores an Gauge into an SNMP message.
IP_SNMP_AGENT_StoreGauge32()	Stores an Gauge32 into an SNMP message.
IP_SNMP_AGENT_StoreInstanceNA()	Stores a Varbind exception into an SNMP message if an instance (index) addressed is not available.
IP_SNMP_AGENT_StoreInteger()	Stores an INTEGER into an SNMP message.
IP_SNMP_AGENT_StoreInteger32()	Stores an INTEGER32 into an SNMP message.
IP_SNMP_AGENT_StoreInteger64()	Stores an Integer64 into an SNMP message.
IP_SNMP_AGENT_StoreIpAddress()	Stores an IpAddress into an SNMP message.
IP_SNMP_AGENT_StoreOctetString()	Stores an octet string into an SNMP message.
IP_SNMP_AGENT_StoreOID()	Stores an OID into an SNMP message.
IP_SNMP_AGENT_StoreOpaque()	Stores an Opaque into an SNMP message.
IP_SNMP_AGENT_StoreTimeTicks()	Stores a 32-bit TimeTick into an SNMP message.
IP_SNMP_AGENT_StoreUnsigned32()	Stores an Unsigned32 into an SNMP message.
IP_SNMP_AGENT_StoreUnsigned64()	Stores an Unsigned64 into an SNMP message.
Message parsing functions	
IP_SNMP_AGENT_ParseBits()	Parses a bitfield out of an SNMP message.
IP_SNMP_AGENT_ParseCounter()	Parses a Counter field out of an SNMP message.
IP_SNMP_AGENT_ParseCounter32()	Parses a Counter32 field out of an SNMP message.

Function	Description
IP_SNMP_AGENT_ParseCounter64()	Parses a Counter64 field out of an SNMP message.
IP_SNMP_AGENT_ParseDouble()	Parses a double field out of an SNMP message.
IP_SNMP_AGENT_ParseFloat()	Parses a float field out of an SNMP message.
IP_SNMP_AGENT_ParseGauge()	Parses a Gauge field out of an SNMP message.
IP_SNMP_AGENT_ParseGauge32()	Parses a Gauge32 field out of an SNMP message.
IP_SNMP_AGENT_ParseInteger()	Parses an INTEGER field out of an SNMP message.
IP_SNMP_AGENT_ParseInteger32()	Parses an INTEGER32 field out of an SNMP message.
IP_SNMP_AGENT_ParseInteger64()	Parses an Integer64 field out of an SNMP message.
IP_SNMP_AGENT_ParseIpAddress()	Parses an IpAddr field (IPv4) out of an SNMP message.
IP_SNMP_AGENT_ParseOctetString()	Parses an OCTET STRING out of an SNMP message.
IP_SNMP_AGENT_ParseOID()	Parses an OID out of an SNMP message.
IP_SNMP_AGENT_ParseOpaque()	Parses an Opaque field out of an SNMP message.
IP_SNMP_AGENT_ParseTimeTicks()	Parses a 32-bit TimeTick field out of an SNMP message.
IP_SNMP_AGENT_ParseUnsigned32()	Parses an Unsigned32 field out of an SNMP message.
IP_SNMP_AGENT_ParseUnsigned64()	Parses an Unsigned64 field out of an SNMP message.
Helper functions	
IP_SNMP_AGENT_DecodeOIDValue()	Parses and decodes an OID value of max (2^32) - 1 into an U32 to work with.
IP_SNMP_AGENT_EncodeOIDValue()	Encodes an OID value of max (2^32) - 1 (U32) into a buffer.
IP_SNMP_AGENT_TRAP_INFORM_SetIPv4AddrPort()	Helper function that sets an IPv4 address in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.
IP_SNMP_AGENT_TRAP_INFORM_SetIPv6AddrPort()	Helper function that sets an IPv6 address in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.
IP_SNMP_AGENT_TRAP_INFORM_SetType()	Helper function that sets the Type structure member in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.
IP_SNMP_AGENT_TRAP_INFORM_SetCommunity()	Helper function that sets the Community structure member in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.
IP_SNMP_AGENT_TRAP_INFORM_SetUser()	Helper function that sets the User structure member in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.

Function	Description
IP_SNMP_AGENT_TRAP_INFORM_SetTimeoutRetries()	Helper function that sets the Timeout and Retries structure members in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.
IP_SNMP_AGENT_TRAP_INFORM_SetMPFlags()	Helper function that sets the Message Processor (MP) flags to use in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.
IP_SNMP_SM_USM_USER_SetEngine()	Helper function that sets the Engine structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.
IP_SNMP_SM_USM_USER_SetUsername()	Helper function that sets the Username structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.
IP_SNMP_SM_USM_USER_SetPerm()	Helper function that sets the permission structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.
IP_SNMP_SM_USM_USER_SetAuthParamsAndKey()	Helper function that sets the AUTH(entication) parameters used for AUTH(entication) handling and the calculated AuthKey structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.
IP_SNMP_SM_USM_USER_SetPrivParamsAndKey()	Helper function that sets the PRIV(acy) parameters used for PRIV(acy) handling and the calculated PrivKey structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.

33.8.1 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);
```

Parameters

Parameter	Description
pCommunity	Pointer to IP_SNMP_AGENT_COMMUNITY memory block.
sCommunity	Community string to add.
Len	Length of the community string without termination.

33.8.2 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);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

33.8.3 IP_SNMP_AGENT_AddInformResponseHook()

Description

Registers a hook that is called upon a state change of an INFORM item waiting for an ACK.

Prototype

```
void IP_SNMP_AGENT_AddInformResponseHook
                                    (IP_SNMP_AGENT_HOOK_ON_INFORM_RESPONSE * pHook,
                                     IP_SNMP_AGENT_pfOnInformResponse        pf);
```

Parameters

Parameter	Description
pHook	Pointer to element of type IP_SNMP_AGENT_HOOK_ON_INFORM_RESPONSE.
pf	Function pointer to callback to hook in.

33.8.4 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);
```

Parameters

Parameter	Description
pTrapInformContext	Pointer to context of message sent.

Additional information

Canceling a message will not overwrite an already set status.

33.8.5 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);
```

Parameters

Parameter	Description
pContext	Pointer to INFORM context that has been used for sending.

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

33.8.6 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.

33.8.7 IP_SNMP_AGENT_Exec()

Description

Executes management tasks.

Prototype

```
U32 IP_SNMP_AGENT_Exec(void);
```

Return value

Time [ms] until the next timeout expires. Returns with 1000 milliseconds/ticks if nothing expires earlier.

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.

33.8.8 IP_SNMP_AGENT_GetMessageType()

Description

Retrieves the message type of the message in the given buffer.

Prototype

```
int IP_SNMP_AGENT_GetMessageType(U8  * pIn,
                                 U32   NumBytesIn,
                                 U8  * pType);
```

Parameters

Parameter	Description
pIn	Pointer to received SNMP message. The buffer needs to be able to get temporarily modified.
NumBytesIn	Length of received SNMP message.
pType	Pointer where to store the parsed PDU message type.

Return value

= 0 O.K.
≠ 0 Error.

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.

Only supports SNMPv1/SNMPv2c messages. For SNMPv3 coexistance use `IP_SNMP_AGENT_ProcessMessage()` directly for SNMPv1/SNMPv2c/SNMPv3 .

33.8.9 IP_SNMP_AGENT_Init()

Description

Initializes the SNMP Agent.

Prototype

```
void IP_SNMP_AGENT_Init(const IP_SNMP_AGENT_API * pAPI);
```

Parameters

Parameter	Description
pAPI	Pointer to SNMP Agent API.

33.8.10 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pUserContext	User specific context passed to callbacks.
pEnterpriseOID	Pointer to enterprise OID sent in TRAP/INFORM.
EnterpriseOIDLen	Length of enterprise OID.
pTrapOID	Pointer to SMIv2 TRAP OID sent as source.
TrapOIDLen	Length of TRAP OID.
pBuffer	Pointer to buffer where to construct the Varbinds to send.
BufferSize	Size of the construct buffer.
AgentAddr	IP addr. of Agent sending the traps. For IPv6 simply 0.

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 inlcuded. 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
IP_SNMP_GENERIC_TRAP_OID_ENTERPRISE_SPECIFIC	1.3.6.1.6.3.1.1.5.7

33.8.11 IP_SNMP_AGENT_ProcessInformResponse()

Description

Processes a response that has been received for a previously sent INFORM message.

Prototype

```
int IP_SNMP_AGENT_ProcessInformResponse(U8  * pIn,
                                        U32   NumBytesIn);
```

Parameters

Parameter	Description
pIn	Pointer to received SNMP message. The buffer needs to be able to get temporarily modified.
NumBytesIn	Length of received SNMP message.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

Only supports SNMPv1/SNMPv2c messages. For SNMPv3 coexistance use `IP_SNMP_AGENT_ProcessMessage()` directly.

33.8.12 IP_SNMP_AGENT_ProcessMessage()

Description

Processes a received message and fills the output buffer with the response for sending back.

Prototype

```
int IP_SNMP_AGENT_ProcessMessage(U8   * pIn,
                                 U32    NumBytesIn,
                                 U8   * pOut,
                                 U32    NumBytesOut,
                                 void * pUserContext);
```

Parameters

Parameter	Description
pIn	Pointer to received SNMP message. The buffer needs to be able to get temporarily modified.
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.

Return value

> 0	Length of response to send.
= 0	No response to send.
< 0	Error.

33.8.13 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);
```

Parameters

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.

Return value

= 0	(TRAP) O.K., pContext and pTrapInformContext can be freed.
< 0	(TRAP) Error (buffer not big enough ?), pContext and pTrapInformContext can be freed.
= 1	(INFORM) O.K., pContext and pTrapInformContext need to be preserved for further usage.
< 0	(INFORM) Error (buffer not big enough ?), pContext and pTrapInformContext need to be preserved for further usage.

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() should be called for the message in question. After this it is safe to free the resources.

33.8.14 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);
```

Parameters

Parameter	Description
pCommunity	Pointer to IP_SNMP_AGENT_COMMUNITY memory block.
pPerm	Pointer to the NULL entry terminated permissions table to use for this community.

33.8.15 IP_SNMP_AGENT_MPV3_Add()

Description

Adds the MessageProcessor (MP) for SNMPv3 messages.

Prototype

```
void IP_SNMP_AGENT_MPV3_Add(const IP_SNMP_AGENT_MPV3_CONFIG * pConfig);
```

Parameters

Parameter	Description
pConfig	Pointer to a configuration of type IP_SNMP_AGENT_MPV3_CONFIG .

Additional information

First the MP for SNMPv3 message handling needs to be added using this routine. Then an SNMPv3 security model needs to be added.

33.8.16 IP_SNMP_AGENT_SetInformReportCallback()

Description

Sets a callback that is executed when information about an SNMPv3 engine are received.

Prototype

```
void IP_SNMP_AGENT_SetInformReportCallback
                                        (IP_SNMP_AGENT_ON_INFORM_REPORT_FUNC * pf);
```

Parameters

Parameter	Description
pf	Callback to execute when Engine information is received.

33.8.17 IP_SNMP_AGENT_SM_USM_Add()

Description

Adds the User-basedSecurityModel (USM) processor for SNMPv3 messages.

Prototype

```
void IP_SNMP_AGENT_SM_USM_Add(const IP_SNMP_AGENT_SM_USM_CONFIG * pConfig);
```

Parameters

Parameter	Description
pConfig	Pointer to a configuration of type IP_SNMP_AGEN-T_SM_USM_CONFIG .

Additional information

The SNMPv3 Messageprocessor (MP) needs to be added before adding any SecurityModel (SM).

33.8.18 IP_SNMP_AGENT_SM_USM_CalcKey()

Description

Calculates the AuthKey or PrivKey for a password and SNMP EngineId.

Prototype

```
int IP_SNMP_AGENT_SM_USM_CalcKey(const IP_SNMP_SM_USM_AUTH_PARAMS * pAuthParams,
                                       U8                         * pBuffer,
                                       unsigned                     BufferSize,
                                 const U8                         * pEngineId,
                                       unsigned                     EngineIdLen,
                                 const U8                         * pPass,
                                       unsigned                     PassLen);
```

Parameters

Parameter	Description
pAuthParams	Pointer to a configuration of type IP_SNMP_SM_USM_AUTH_PARAMS .
pBuffer	Pointer where to store the calculated key. The key/digest length depends on the selected hash algorithm of pAuthParams .
BufferSize	Available buffer size.
pEngineId	Pointer to the SNMP EngineId for which to calculate a key.
EngineIdLen	Length of the EngineId.
pPass	Pointer to the password.
PassLen	Length of the password.

Return value

= 0 O.K.
≠ 0 Error, buffer too small for hash algorithm digest ?

Additional information

SNMP uses AUTH and PRIV keys that are calculated per EngineId. This means that one and the same password can not be used with different EngineIds. The username however is not part of the calculation which means that if multiple users share the same password on the same EngineId, this becomes visible in the user table as they will have the same AUTH or PRIVACY key.

Depending on the overall CPU speed, calculating the key from a given password on the fly when needed might easily exceed the timeouts of request. On systems that do not come with enough RAM to populate an SNMP user table on the fly, most likely the CPU is also not fast enough to calculate the key on the fly when needed. Therefore the user table is created upfront and is then installed for immediate lookups of the key for a user.

33.8.19 IP_SNMP_AGENT_SM_USM_SetUserTable()

Description

Sets the SNMPv3 user table.

Prototype

```
int IP_SNMP_AGENT_SM_USM_SetUserTable
                              (const IP_SNMP_SM_USM_USER_TABLE_ENTRY * pUserTable,
                                     U8                                NumEntries);
```

Parameters

Parameter	Description
pUserTable	Pointer to the first entry of the user table of type IP_SNMP_SM_USM_USER_TABLE_ENTRY .
NumEntries	Number of entries at pUserTable .

Return value

= 0 O.K.
≠ 0 Sanity check failed for usernames using more than 32 characters or something else. Please refer to the debug warning outputs for more details.

33.8.20 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);
```

Parameters

Parameter	Description
pAPI	Pointer to information and callback structure of type IP_SNMP_AGENT_MIB2_INTERFACES_API.

Return value

= 0 O.K.
≠ 0 Error.

33.8.21 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);
```

Parameters

Parameter	Description
pAPI	Pointer to information and callback structure of type IP_SNMP_AGENT_MIB2_SYSTEM_API.

Return value

= 0 O.K.
≠ 0 Error.

33.8.22 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

= 0	O.K.
≠ 0	Error.

33.8.23 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.

Return value

= 0 O.K.
≠ 0 Error.

33.8.24 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.

Return value

= 0 O.K.
≠ 0 Error (buffer not big enough ?).

33.8.25 IP_SNMP_AGENT_StoreBits()

Description

Stores a bitfield into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreBits(      IP_SNMP_AGENT_CONTEXT * pContext,
                            const U8                    * pData,
                                  U32                     NumBytes);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pData	Pointer to bitfield.
NumBytes	Length of bitfield in bytes.

Return value

= 0 O.K.
≠ 0 Error.

33.8.26 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Counter to store.

Return value

= 0 O.K.
≠ 0 Error.

33.8.27 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Counter32 to store.

Return value

= 0 O.K.
≠ 0 Error.

33.8.28 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Counter64 to store.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES = 1.

33.8.29 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,
                                             ...);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

33.8.30 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Double to store.

Return value

= 0 O.K.
≠ 0 Error.

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`.

33.8.31 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Float to store.

Return value

= 0 O.K.
< 0 Error.

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.

33.8.32 IP_SNMP_AGENT_StoreGauge()

Description

Stores an Gauge into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreGauge(IP_SNMP_AGENT_CONTEXT * pContext,
                             U32                     v);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Gauge to store.

Return value

= 0 O.K.
≠ 0 Error.

33.8.33 IP_SNMP_AGENT_StoreGauge32()

Description

Stores an Gauge32 into an SNMP message.

Prototype

```
int IP_SNMP_AGENT_StoreGauge32(IP_SNMP_AGENT_CONTEXT * pContext,
                               U32                     v);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Gauge32 to store.

Return value

= 0 O.K.
≠ 0 Error.

33.8.34 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.

Return value

= 0 O.K.
≠ 0 Error.

33.8.35 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	INTEGER to store.

Return value

= 0 O.K.
≠ 0 Error.

33.8.36 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	INTEGER32 to store.

Return value

= 0 O.K.
≠ 0 Error.

33.8.37 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Integer64 to store.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES = 1.

33.8.38 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
IpAddress	IPv4 addr. as U32 in host order to store.

Return value

= 0 O.K.
≠ 0 Error.

33.8.39 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pData	Pointer to octet string to store.
NumBytes	Length of octet string.

Return value

= 0 O.K.
≠ 0 Error.

33.8.40 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);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

33.8.41 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pData	Pointer to Opaque data to store.
NumBytes	Size of Opaque data.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.42 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	TimeTicks to store.

Return value

= 0 O.K.
≠ 0 Error.

33.8.43 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Unsigned32 to store.

Return value

= 0 O.K.
≠ 0 Error.

33.8.44 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
v	Unsigned64 to store.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES = 1.

33.8.45 IP_SNMP_AGENT_ParseBits()

Description

Parses a bitfield out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseBits(      IP_SNMP_AGENT_CONTEXT  * pContext,
                            const U8                   ** ppData,
                                  U32                   * pLen);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

This function expects that the type field has not been eaten out of the buffer.

33.8.46 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                   * pCounter);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pCounter	Pointer where to store the parsed Counter.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.47 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pCounter32	Pointer where to store the parsed Counter32.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.48 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pCounter64	Pointer where to store the parsed Counter64.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.49 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pDouble	Pointer where to store the parsed double.

Return value

= 0 O.K.
≠ 0 Error.

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`.

33.8.50 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pFloat	Pointer where to store the parsed float.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.51 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pUnsigned32	Pointer where to store the parsed Gauge.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

By design an 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.

33.8.52 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pUnsigned32	Pointer where to store the parsed Gauge32.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

By design an 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.

33.8.53 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pInteger	Pointer where to store the parsed INTEGER.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.54 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                   * pInteger32);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pInteger32	Pointer where to store the parsed INTEGER32.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.55 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pInteger64	Pointer where to store the parsed Integer64.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.56 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);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

This function expects that the type field has not been eaten out of the buffer.

33.8.57 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,
                                   const U8                    ** ppData,
                                         U32                    * pLen);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

This function expects that the type field has not been eaten out of the buffer.

33.8.58 IP_SNMP_AGENT_ParseOID()

Description

Parses an OID out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseOID(      IP_SNMP_AGENT_CONTEXT  * pContext,
                           const U8                   ** ppData,
                                 U32                   * pLen);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

This function expects that the type field has not been eaten out of the buffer.

33.8.59 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,
                              const U8                    ** ppData,
                                    U32                    * pLen);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

This function expects that the type field has not been eaten out of the buffer.

33.8.60 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pTimeTicks	Pointer where to store the parsed TimeTicks.

Return value

= 0 O.K.
≠ 0 Error.

Additional information

This function expects that the type field has not been eaten out of the buffer.

33.8.61 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);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pUnsigned32	Pointer where to store the parsed Unsigned32.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.62 IP_SNMP_AGENT_ParseUnsigned64()

Description

Parses an Unsigned64 field out of an SNMP message.

Prototype

```
int IP_SNMP_AGENT_ParseUnsigned64(IP_SNMP_AGENT_CONTEXT * pContext,
                                  U64                   * pUnsigned64);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pUnsigned64	Pointer where to store the parsed Unsigned64.

Return value

= 0 O.K.
≠ 0 Error.

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 eaten out of the buffer.

Can only be used when IP_SNMP_AGENT_SUPPORT_64_BIT_TYPES = 1.

33.8.63 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);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

33.8.64 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);
```

Parameters

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.

Return value

= 0 O.K.
≠ 0 Error.

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.

33.8.65 IP_SNMP_AGENT_TRAP_INFORM_SetIPv4AddrPort()

Description

Helper function that sets an IPv4 address in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.

Prototype

```
void IP_SNMP_AGENT_TRAP_INFORM_SetIPv4AddrPort
                                (IP_SNMP_AGENT_TRAP_INFORM_CONTEXT * pContext,
                                 U32                                 IPAddr,
                                 U16                                 Port,
                                 U16                                 DiscoverPort);
```

Parameters

Parameter	Description
pContext	Pointer to TRAP/INFORM context of type IP_SNMP_AGENT_TRAP_INFORM_CONTEXT .
IPAddr	IPv4 address where to send the TRAP/INFORM message in host order.
Port	UDP port to send to in host order. Typically 162.
DiscoverPort	UDP port to use for SNMPv3 Engine discovery in host order. Typically 161. Can be 0 if not using Engine discovery.

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_AGENT_TRAP_INFORM_CONTEXT structure to be moved around for best memory efficiency when extending the structure in the future.

33.8.66 IP_SNMP_AGENT_TRAP_INFORM_SetIPv6AddrPort()

Description

Helper function that sets an IPv6 address in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.

Prototype

```
void IP_SNMP_AGENT_TRAP_INFORM_SetIPv6AddrPort
                                (IP_SNMP_AGENT_TRAP_INFORM_CONTEXT * pContext,
                                 U8                                * pIPAddr,
                                 U16                                 Port,
                                 U16                                 DiscoverPort);
```

Parameters

Parameter	Description
pContext	Pointer to TRAP/INFORM context of type IP_SNMP_AGENT_TRAP_INFORM_CONTEXT .
pIPAddr	Pointer to the IPv6 address where to send the TRAP/INFORM message.
Port	UDP port to send to in host order. Typically 162.
DiscoverPort	UDP port to use for SNMPv3 Engine discovery in host order. Typically 161. Can be 0 if not using Engine discovery.

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_AGENT_TRAP_INFORM_CONTEXT structure to be moved around for best memory efficiency when extending the structure in the future.

33.8.67 IP_SNMP_AGENT_TRAP_INFORM_SetType()

Description

Helper function that sets the Type structure member in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.

Prototype

```
void IP_SNMP_AGENT_TRAP_INFORM_SetType
                                    (IP_SNMP_AGENT_TRAP_INFORM_CONTEXT * pContext,
                                     U8                                  Type);
```

Parameters

Parameter	Description
pContext	Pointer to TRAP/INFORM context of type IP_SNMP_AGENT_TRAP_INFORM_CONTEXT .
Type	TRAP/INFORM message type to send: • IP_SNMP_PDU_TYPE_TRAPV1 : Send an SNMPv1 TRAP. • IP_SNMP_PDU_TYPE_TRAPV2 : Send an SNMPv2c TRAP (also SNMPv3). • IP_SNMP_PDU_TYPE_INFORMV2: Send an SNMPv2c INFORM (also SNMPv3).

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_AGENT_TRAP_INFORM_CONTEXT structure to be moved around for best memory efficiency when extending the structure in the future.

SNMPv3 TRAP/INFORM messages use the same PDUs as SNMPv2c TRAP/INFORM messages. To send SNMPv2 TRAP/INFORM messages a community needs to be set using IP_SNMP_AGENT_TRAP_INFORM_SetCommunity() . If no community is set SNMPv3 TRAP/INFORM messages are sent.

33.8.68 IP_SNMP_AGENT_TRAP_INFORM_SetCommunity()

Description

Helper function that sets the Community structure member in an `IP_SNMP_AGENT_TRAP_INFORM_CONTEXT` context.

Prototype

```
void IP_SNMP_AGENT_TRAP_INFORM_SetCommunity
                                  (IP_SNMP_AGENT_TRAP_INFORM_CONTEXT * pContext,
                                   IP_SNMP_AGENT_COMMUNITY           * pCommunity);
```

Parameters

Parameter	Description
`pContext`	Pointer to TRAP/INFORM context of type `IP_SNMP_AGENT_TRAP_INFORM_CONTEXT` .
`pCommunity`	Pointer to community handle of type `IP_SNMP_AGENT_COMMUNITY` .

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the `IP_SNMP_AGENT_TRAP_INFORM_CONTEXT` structure to be moved around for best memory efficiency when extending the structure in the future.

SNMPv3 TRAP/INFORM messages use the same PDUs as SNMPv1/SNMPv2c TRAP/INFORM messages. To send SNMPv1/SNMPv2 TRAP/INFORM messages a community needs to be set. If no community is set this generates an SNMPv3 TRAP/INFORM messages instead.

33.8.69 IP_SNMP_AGENT_TRAP_INFORM_SetUser()

Description

Helper function that sets the User structure member in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.

Prototype

```
void IP_SNMP_AGENT_TRAP_INFORM_SetUser
                                    (IP_SNMP_AGENT_TRAP_INFORM_CONTEXT * pContext,
                                     IP_SNMP_SM_USM_USER_TABLE_ENTRY   * pUser);
```

Parameters

Parameter	Description
pContext	Pointer to TRAP/INFORM context of type IP_SNMP_AGENT_TRAP_INFORM_CONTEXT .
pUser	Pointer to User handle of type IP_SNMP_SM_USM_USER_TABLE_ENTRY .

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_AGENT_TRAP_INFORM_CONTEXT structure to be moved around for best memory efficiency when extending the structure in the future.

SNMPv3 TRAP/INFORM messages use the same PDUs as SNMPv1/SNMPv2c TRAP/INFORM messages. To send SNMPv1/SNMPv2 TRAP/INFORM messages a community needs to be set. If no community is set this generates an SNMPv3 TRAP/INFORM messages instead.

33.8.70 IP_SNMP_AGENT_TRAP_INFORM_SetTimeoutRetries()

Description

Helper function that sets the Timeout and Retries structure members in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.

Prototype

```
void IP_SNMP_AGENT_TRAP_INFORM_SetTimeoutRetries
                                    (IP_SNMP_AGENT_TRAP_INFORM_CONTEXT * pContext,
                                     U32                                 Timeout,
                                     U8                                  Retries);
```

Parameters

Parameter	Description
pContext	Pointer to TRAP/INFORM context of type IP_SNMP_AGENT_TRAP_INFORM_CONTEXT .
Timeout	INFORM timeout [ms] of each message sent.
Retries	Number of INFORM retries to send.

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_AGENT_TRAP_INFORM_CONTEXT structure to be moved around for best memory efficiency when extending the structure in the future.

The Timeout and Retries parameters are typically only used when sending INFORM messages. When the peer EngineId shall be discovered the Retries value is the combined number of retries for discovering the peer EngineId and receiving a response for the INFORM message.

33.8.71 IP_SNMP_AGENT_TRAP_INFORM_SetMPFlags()

Description

Helper function that sets the Message Processor (MP) flags to use in an IP_SNMP_AGENT_TRAP_INFORM_CONTEXT context.

Prototype

```
void IP_SNMP_AGENT_TRAP_INFORM_SetMPFlags
                                    (IP_SNMP_AGENT_TRAP_INFORM_CONTEXT * pContext,
                                     U8                                  MPFlags);
```

Parameters

Parameter	Description
pContext	Pointer to TRAP/INFORM context of type IP_SNMP_AGENT_TRAP_INFORM_CONTEXT .
MPFlags	OR-combination of IP_SNMPV3_MSG_FLAG_* to use. Valid flags are: • IP_SNMPV3_MSG_FLAG_AUTH_MASK: If the user has AUTH(thentication) parameters and shall use them. • IP_SNMPV3_MSG_FLAG_PRIV_MASK: If the user has PRIV(acy) parameters and shall use them (automatically sets IP_SNMPV3_MSG_FLAG_AUTH_MASK as well).

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_AGENT_TRAP_INFORM_CONTEXT structure to be moved around for best memory efficiency when extending the structure in the future.

The Timeout and Retries parameters are only required for sending INFORM messages.

33.8.72 IP_SNMP_SM_USM_USER_SetEngine()

Description

Helper function that sets the Engine structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.

Prototype

```
void IP_SNMP_SM_USM_USER_SetEngine(IP_SNMP_SM_USM_USER_TABLE_ENTRY * pEntry,
                                   IP_SNMP_SM_USM_ENGINE_ENTRY     * pEngine);
```

Parameters

Parameter	Description
pEntry	Pointer to user table entry of type IP_SNMP_SM_USM_USER_TABLE_ENTRY .
pEngine	Pointer to the Engine to use for this user of type IP_SNMP_SM_USM_ENGINE_ENTRY .

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_SM_USM_USER_TABLE_ENTRY structure to be moved around for best memory efficiency when extending the structure in the future.

33.8.73 IP_SNMP_SM_USM_USER_SetUsername()

Description

Helper function that sets the Username structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.

Prototype

```
void IP_SNMP_SM_USM_USER_SetUsername
                                    (IP_SNMP_SM_USM_USER_TABLE_ENTRY * pEntry,
                                     U8                              * pUsername,
                                     U8                                UsernameLen);
```

Parameters

Parameter	Description
pEntry	Pointer to user table entry of type IP_SNMP_SM_USM_USER_TABLE_ENTRY .
pUsername	Pointer to the Username to set (without string termination).
UsernameLen	Length of the Username at pUsername .

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_SM_USM_USER_TABLE_ENTRY structure to be moved around for best memory efficiency when extending the structure in the future.

33.8.74 IP_SNMP_SM_USM_USER_SetPerm()

Description

Helper function that sets the permission structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.

Prototype

```
void IP_SNMP_SM_USM_USER_SetPerm(IP_SNMP_SM_USM_USER_TABLE_ENTRY * pEntry,
                                 IP_SNMP_AGENT_PERM              * pPerm);
```

Parameters

Parameter	Description
pEntry	Pointer to user table entry of type IP_SNMP_SM_USM_USER_TABLE_ENTRY .
pPerm	Pointer to the NULL entry terminated permissions table to set.

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_SM_USM_USER_TABLE_ENTRY structure to be moved around for best memory efficiency when extending the structure in the future.

33.8.75 IP_SNMP_SM_USM_USER_SetAuthParamsAndKey()

Description

Helper function that sets the AUTH(thentication) parameters used for AUTH(entication) handling and the calculated AuthKey structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.

Prototype

```
void IP_SNMP_SM_USM_USER_SetAuthParamsAndKey
                                    (IP_SNMP_SM_USM_USER_TABLE_ENTRY * pEntry,
                                     IP_SNMP_SM_USM_AUTH_PARAMS      * pAuthParams,
                                     U8                              * pAuthKey);
```

Parameters

Parameter	Description
pEntry	Pointer to user table entry of type IP_SNMP_SM_USM_USER_TABLE_ENTRY .
pAuthParams	Pointer to a configuration of type IP_SNMP_SM_USM_AUTH_PARAMS .
pAuthKey	Pointer to the calculated AuthKey.

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_SM_USM_USER_TABLE_ENTRY structure to be moved around for best memory efficiency when extending the structure in the future.

A call to this function can be omitted if the entry created shall be of type “noAuthNoPriv”.

33.8.76 IP_SNMP_SM_USM_USER_SetPrivParamsAndKey()

Description

Helper function that sets the PRIV(acy) parameters used for PRIV(acy) handling and the calculated PrivKey structure member in an IP_SNMP_SM_USM_USER_TABLE_ENTRY entry.

Prototype

```
void IP_SNMP_SM_USM_USER_SetPrivParamsAndKey
                                    (IP_SNMP_SM_USM_USER_TABLE_ENTRY * pEntry,
                                     IP_SNMP_SM_USM_PRIV_PARAMS      * pPrivParams,
                                     U8                              * pPrivKey);
```

Parameters

Parameter	Description
pEntry	Pointer to user table entry of type IP_SNMP_SM_USM_USER_TABLE_ENTRY .
pPrivParams	Pointer to a configuration of type IP_SNMP_SM_USM_PRIV_PARAMS .
pPrivKey	Pointer to the calculated PrivKey.

Additional information

The purpose of this helper function is to provide a persistent API while allowing the members of the IP_SNMP_SM_USM_USER_TABLE_ENTRY structure to be moved around for best memory efficiency when extending the structure in the future.

A call to this function can be omitted if the entry created shall be of type “noAuthNoPriv”.

33.9 Data structures

33.9.1 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 IP_SNMP_AGENT_Init().
pfDeInit	Callback for deinitialization called from IP_SNMP_AGENT_DeInit().
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).

33.9.2 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.

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(). 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.

33.9.3 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.

33.9.4 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 emNet is shipped with the SNMP Agent in the file `IP_SNMP_AGENT_MIB2_INTERFACES_emNet.c` . The members of the structure are based on the SNMP structure of MIB-II interface counters.

To enable statistic counters in emNet to provide the SNMP MIB-II interfaces counters with valid values please enable `IP_SUPPORT_STATS_IFACE`.

33.9.5 IP_SNMP_HASH_INIT_FUNC

Description

Returns/initializes a fresh hash context.

Type definition

```
typedef void * IP_SNMP_HASH_INIT_FUNC(void);
```

Return value

Initialized hash context.

Additional information

Calculating hashes is done from a task that uses the API lock. Therefore it is typically sufficient to use a single static hash context.

For the moment the routine is not expected to fail and return a `NULL` pointer.

33.9.6 IP_SNMP_HASH_ADD_FUNC

Description

Adds data to the hash calculation.

Type definition

```
typedef void IP_SNMP_HASH_ADD_FUNC(      void   * pContext,
                                   const U8     * pInput,
                                         unsigned InputLen);
```

Parameters

Parameter	Description
pContext	Pointer to hash context returned from init callback.
pInput	Pointer to data to add.
InputLen	Length of the data to add from pInput .

33.9.7 IP_SNMP_HASH_FINAL_FUNC

Description

Finalizes the hash calculation and returns the digest.

Type definition

```
typedef void IP_SNMP_HASH_FINAL_FUNC(void   * pContext,
                                      U8     * pDigest,
                                      unsigned DigestLen);
```

Parameters

Parameter	Description
pContext	Pointer to hash context returned from init callback.
pDigest	Pointer where to store the result.
DigestLen	Maximum size of the buffer where to store the result.

33.9.8 IP_SNMP_HASH_API

Description

Hash API for the User-basedSecurityModel (USM) AUTH(entication) of a user.

Type definition

```
typedef struct {
  IP_SNMP_HASH_INIT_FUNC  * pfInit;
  IP_SNMP_HASH_ADD_FUNC   * pfAdd;
  IP_SNMP_HASH_FINAL_FUNC * pfFinal;
} IP_SNMP_HASH_API;
```

Structure members

Member	Description
pfInit	Calllback to allocate a fresh hash algorithm context.
pfAdd	Calllback to add more data into the hash algorithm.
pfFinal	Calllback to finalize hashing and return the digest.

33.9.9 IP_SNMP_SM_USM_AUTH_PARAMS

Description

Configuration parameters for the User-basedSecurityModel (USM) authentication for a user.

Type definition

```
typedef struct {
  const IP_SNMP_SM_USM_AUTH_API * pAuthAPI;
  const IP_SNMP_HASH_API        * pHashAPI;
} IP_SNMP_SM_USM_AUTH_PARAMS;
```

Structure members

Member	Description
pAuthAPI	Pointer to the AUTH(entication) specific handling API.
pHashAPI	Pointer to the hash API to use.

33.9.10 IP_SNMP_SM_USM_PRIV_API_EXEC_FUNC

Description

Executes the PRIV(acy) specific cipher handling.

Type definition

```
typedef int IP_SNMP_SM_USM_PRIV_API_EXEC_FUNC(IP_SNMP_AGENT_CONTEXT * pContext,
                                               U8                    * pData,
                                               unsigned                NumBytes,
                                               unsigned                SaltLen,
                                               IP_SNMP_CIPHER_DIR      Direction);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pData	Pointer to the data to decrypt or encrypt in-place.
NumBytes	Number of bytes to decrypt/encrypt.
SaltLen	Length of the value of the “msgPrivacyParameters” field.
Direction	Decrypt or encrypt direction of type IP_SNMP_CIPHER_DIR . • IP_SNMP_CIPHER_DIR_DECRYPT • IP_SNMP_CIPHER_DIR_ENCRYPT

Return value

= 0 O.K.
< 0 Error (not enough bytes in buffer?)

33.9.11 IP_SNMP_SM_USM_PRIV_API

Description

Cipher specific driver-like API for the User-basedSecurityModel (USM) PRIV(acy) of a user.

Type definition

```
typedef struct {
  IP_SNMP_SM_USM_PRIV_API_EXEC_FUNC * pfExec;
  U8                                  BlockLen;
} IP_SNMP_SM_USM_PRIV_API;
```

Structure members

Member	Description
pfExec	Callback executing the cipher specific PRIV(acy) handling.
BlockLen	Length of each individual ciphertext block.

33.9.12 IP_SNMP_CIPHER_INIT_FUNC

Description

Returns/initializes a fresh cipher context for a decrypt or encrypt operation.

Type definition

```
typedef void * IP_SNMP_CIPHER_INIT_FUNC(const U8               * pKey,
                                         unsigned                KeyLen,
                                         IP_SNMP_CIPHER_DIR Direction);
```

Parameters

Parameter	Description
pKey	Pointer to the cipher key to use. Its length is determined by the PRIV(acy) cipher selected via the user table.
KeyLen	Length of the key at pKey .
Direction	Decrypt or encrypt direction of type IP_SNMP_CIPHER_DIR . • IP_SNMP_CIPHER_DIR_DECRYPT • IP_SNMP_CIPHER_DIR_ENCRYPT

Return value

Initialized cipher context.

Additional information

Decrypt/encrypt is done from a task that uses the API lock. Therefore it is typically sufficient to use a single static cipher context.

For the moment the routine is not expected to fail and return a NULL pointer.

33.9.13 IP_SNMP_CIPHER_EXEC_FUNC

Description

Decrypts/encrypts data.

Type definition

```
typedef void IP_SNMP_CIPHER_EXEC_FUNC(      void             * pContext,
                                            U8               * pOutput,
                                      const U8               * pInput,
                                            unsigned           InputLen,
                                            U8               * pIV,
                                            IP_SNMP_CIPHER_DIR Direction);
```

Parameters

Parameter	Description
pContext	Pointer to cipher context returned from init callback.
pOutput	Pointer where to store the decrypted output.
pInput	Pointer to the encrypted input.
InputLen	Length of the data to decrypt from pInput .
pIV	Pointer to the IV (InitializationVector) to use. The size of the IV is determined by the PRIV(acy) cipher selected via the user table.
Direction	Decrypt or encrypt direction of type IP_SNMP_CIPHER_DIR . • IP_SNMP_CIPHER_DIR_DECRYPT • IP_SNMP_CIPHER_DIR_ENCRYPT

33.9.14 IP_SNMP_CIPHER_FINAL_FUNC

Description

Finalizes the decrypt or encrypt operation.

Type definition

```
typedef void IP_SNMP_CIPHER_FINAL_FUNC(void             * pContext,
                                        IP_SNMP_CIPHER_DIR Direction);
```

Parameters

Parameter	Description
pContext	Pointer to hash context returned from init callback.
Direction	Decrypt or encrypt direction of type IP_SNMP_CIPHER_DIR . • IP_SNMP_CIPHER_DIR_DECRYPT • IP_SNMP_CIPHER_DIR_ENCRYPT

Additional information

This callback can be used to free resources allocated during init or to kill any security related leftovers from the cipher operation.

33.9.15 IP_SNMP_CIPHER_API

Description

Cipher API for the User-basedSecurityModel (USM) PRIV(acy) of a user.

Type definition

```
typedef struct {
  IP_SNMP_CIPHER_INIT_FUNC  * pfInit;
  IP_SNMP_CIPHER_EXEC_FUNC  * pfExec;
  IP_SNMP_CIPHER_FINAL_FUNC * pfFinal;
} IP_SNMP_CIPHER_API;
```

Structure members

Member	Description
pfInit	Calllback to allocate a fresh cipher algorithm context.
pfExec	Calllback to decrypt/encrypt data.
pfFinal	Calllback to finalize cipher operations and free resources.

33.9.16 IP_SNMP_SM_USM_PRIV_PARAMS

Description

Configuration parameters for the User-basedSecurityModel (USM) data encryption for a user.

Type definition

```
typedef struct {
  const IP_SNMP_SM_USM_PRIV_API * pPrivAPI;
  const IP_SNMP_CIPHER_API      * pCipherAPI;
} IP_SNMP_SM_USM_PRIV_PARAMS;
```

Structure members

Member	Description
pPrivAPI	Pointer to the PRIV(acy) specific handling API.
pCipherAPI	Pointer to the cipher API to use.

33.9.17 IP_SNMP_SM_USM_ENGINE_ENTRY

Description

Information related to an SNMP Engine.

Type definition

```
typedef struct {
  const U8 * pEngineId;
  I32        EngineBoots;
  I32        EngineTime;
  U8         EngineIdLen;
} IP_SNMP_SM_USM_ENGINE_ENTRY;
```

Structure members

Member	Description
pEngineId	Pointer to an EngineId.
EngineBoots	Number of how often the SNMP engine has been "booted". Ideally this value is increased for each time the hardware boots or SNMP is started. However, EngineBoots shall also be incremented once EngineTime reaches its I32 maximum of 2147483647 seconds, then resetting EngineTime back to 0 again.
EngineTime	Seconds since the SNMP engine has "booted". Once the I32 maximum of 2147483647 seconds is reached, EngineBoots shall be incremented and EngineTime starts from 0 again.
EngineIdLen	Length of the EngineId.

Additional information

An SNMP(v3) Engine not only has an EngineId but also has some parameters that need to be maintained such as the EngineBoots and EngineTime parameters. Some of these parameters might even be actively learned from peer Engines when receiving REPORT messages.

If an SNMP Engine entry is used for the local Engine the EngineTime of this entry needs to be periodically updated by the application. The EngineTime should be updated once every second or at least before the previous "EngineTime + IP_SNMP_AGENT_SM_USM_CONFIG.Timeout" expires.

The SNMPv3 USM "msgAuthoritativeEngineBoots" and "msgAuthoritativeEngineTime" values do not necessarily reflect an actual 1:1 time of the system since booting. Their value is only exchanged between two SNMP entities once AUTH(entication) has succeeded. If the time of an SNMP engine is unknown or outside the time window, the time might need to be retrieved in a separate request. The initiator/client can maintain the discovered values on its own and try to directly send more messages using its own maintained values without having to discover the engine time to use again and again.

The parameters EngineBoots and EngineTime are meant to be stored in non-volatile memory when the SNMP Agent is shut down and be restored when starting again. This procedure does not even need to increase the EngineBoots necessarily. This only makes sense if SNMP is started again before the timeout expires in which another entity might be sending further mesages.

The EngineBoots and EngineTime values do not have to be strictly maintained by the application. Their purpose is to prevent replay attacks of messages by limiting the time window in which they can be utilized. It should also be perfectly fine to use randomized start values for these parameters each time as this will typically only lead to having to discover the engine time again with an additional message while also needing to successfully AUTH(enticate) again for these values to be included in the response.

33.9.18 IP_SNMP_AGENT_SM_USM_CONFIG

Description

Used to configure the User-basedSecurityModel (USM).

Type definition

```
typedef struct {
  const IP_SNMP_SM_USM_ENGINE_ENTRY * pLocalEngine;
  unsigned                            Timeout;
} IP_SNMP_AGENT_SM_USM_CONFIG;
```

Structure members

Member	Description
pLocalEngine	Pointer to the local SNMP Engine to use.
Timeout	Timeout in seconds, relative to the EngineTime in which a received message is valid. The default according to RFC 3414 is 150 seconds, which means that messages are in the time window if they use NOW +- Timeout .

33.9.19 IP_SNMP_SM_USM_USER_TABLE_ENTRY

Description

The SNMPv3 user access is managed by a table/array of IP_SNMP_SM_USM_USER_TABLE_ENTRY items that allow to set different combinations of "noAuthNoPriv", "authNoPriv" and "authPriv" along with the different hash and encryption algorithms.

Type definition

```
typedef struct {
  const IP_SNMP_SM_USM_ENGINE_ENTRY * pEngine;
  const U8                          * pUsername;
  const IP_SNMP_AGENT_PERM          * pPerm;
  const IP_SNMP_SM_USM_AUTH_PARAMS  * pAuthParams;
  const U8                          * pAuthKey;
  const IP_SNMP_SM_USM_PRIV_PARAMS  * pPrivParams;
  const U8                          * pPrivKey;
  U8                                  UsernameLen;
} IP_SNMP_SM_USM_USER_TABLE_ENTRY;
```

Structure members

Member	Description
pEngine	Pointer to the authorizational Engine that this user entry belongs to.
pUsername	Pointer to the Username (without string termination).
pPerm	Pointer to the permissions table of type IP_SNMP_AGENT_PERM .
pAuthParams	Pointer to an AUTHentication configuration of type IP_SNMP_SM_USM_AUTH_PARAMS . Can be NULL to create a "noAuthNoPriv" type entry.
pAuthKey	Pointer to a calculated AuthKey for the EngineId at pEngineId . The length of the AuthKey is determined by the digest length of pAuthParams . Can be NULL if pAuthParams is NULL .
pPrivParams	Pointer to a PRIVacy configuration of type IP_SNMP_SM_USM_PRIV_PARAMS .
pPrivKey	Pointer to a calculated PrivKey for the EngineId at pEngineId . The length of the PrivKey is determined by the digest length of pAuthParams as the PRIV key is also calculated based on the hash algorithm used for AUTH . Can be NULL if pPrivParams is NULL .
UsernameLen	Length of the Username at pUsername .

Additional information

The user table can be constructed directly by using either the fields in order as they are or by using "Designated Initializers" (initialization with a structs member name in form of ".<StructMember≥<Value>"). Members of this structure might change their order of appearance in the future to allow for memory efficient extension of the structure and user table. For this reason structure members should only be initialized by either using "Designated Initializers" for a ROM/const placement for example or by using the IP_SNMP_SM_USM_USER_* API.

When extending this structure in the future, structure members are expected to move in patterns that will be easily recognizable as they lead to compile errors. In case of doubt (when directly interacting based on fixed order initialization), the application should check the size of the structure and compare it to a previously known size of this structure and raise an error if the size (and most likely order of members) has changed.

To support “view-based” permissions, based on the security level achieved (“noAuthNoPriv”, “authNoPriv” or “authPriv”), a username can be added multiple times for the same EngineId with different parameters being available/set or not set. The selection is done using a perfect match, which means that no entry with a higher security level is used to handle a lower security level request.

When creating the user table as array of `IP_SNMP_SM_USM_USER_TABLE_ENTRY` placeholder entries can be used by settings these entries with “`pEngine = NULL`”. This can be used to allocate contiguous space once and providing space for up to that many entries without having to reallocate the memory when adding or removing entries.

33.9.20 IP_SNMP_USM_ENGINE_INFO

Description

Provides information about a peer SNMPv3 `Engine` that can be used to maintain the list of Engines and their parameters.

Type definition

```
typedef struct {
  IP_SNMP_SM_USM_ENGINE_ENTRY  Engine;
} IP_SNMP_USM_ENGINE_INFO;
```

Structure members

Member	Description
`Engine`	Pointer to `Engine` information of type `IP_SNMP_SM_USM_ENGINE_ENTRY` .

33.9.21 IP_SNMP_AGENT_MPV3_CONFIG

Description

Used to configure the MessageProcessor (MP) for SNMPv3.

Type definition

```
typedef struct {
  I32  MaxSize;
} IP_SNMP_AGENT_MPV3_CONFIG;
```

Structure members

Member	Description
MaxSize	Value to use for the "msgMaxSize" field in SNMPv3 messages. This field describes the maxmimum size of a "ScopedPDU" that can be received without the SNMP headers for various layers preceeding it. This value can not be simply calculated as the header size might differ due to fields of variable length such as the "msgAuthoritativeEngineID" field. The typical Ethernet limit for IPv4 UDP payload is around 1472 bytes for the "ScopedPDU" plus SNMP headers. A "good" value used by other implementations is 1400 bytes.

33.9.22 IP_SNMP_AGENT_ON_INFORM_REPORT_FUNC

Description

Callback executed whenever a REPORT with new information about an SNMPv3 Engine is received for a pending INFORM.

Type definition

```
typedef int (IP_SNMP_AGENT_ON_INFORM_REPORT_FUNC)
                              (IP_SNMP_AGENT_CONTEXT             * pContext,
                               IP_SNMP_AGENT_TRAP_INFORM_CONTEXT * pInformContext,
                               void                              * pUserContext,
                               IP_SNMP_USM_ENGINE_INFO           * pInfo);
```

Parameters

Parameter	Description
pContext	Pointer to an SNMP Agent context.
pInformContext	Pointer to the INFORM context of type IP_SNMP_AGENT_TRAP_INFORM_CONTEXT for which new Engine information have been discovered.
pUserContext	User specific context passed to the process message API.
pInfo	Pointer to information received about an Engine.

Return value

= 0 Retry to send the INFORMs when returning from the callback.
< 0 Do not access pInformContext when returning from the callback (removed?).

Additional information

This callback gets executed when a REPORT message with (new) SNMPv3 Engine information for a yet to be sent INFORM is received. This is typically the case when sending an INFORM while only knowing the IP address of the receiving Manager but not the EngineId. Another case where a REPORT is received is when the peer Engine time window was missed. In this case the REPORT lets us know the current time of the peer Engine and we have to send the INFORM again after updating our information about the peer Engine time which prevents replay attacks with old messages if the AUTH(thorization) security level is used.

Once new Engine information is received the Engine table maintained by the application should be updated and the INFORM should either be resent immediately from within this callback or is resent by the retry mechanism for INFORM messages.

Once initial EngineBoots and EngineTime values have been discovered for an Engine they can be maintained locally by the application to prevent the discover part being necessary if the authoritative Engine is happy with what we send. In the worst case we will receive a REPORT by the peer Engine telling us the latest information.

33.10 Resource usage (SNMPv2c)

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the SNMP Agent for SNMPv2c only support 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.

Configuration used

```
#define IP_SNMP_AGENT_WORK_BUFFER  64
```

33.10.1 ROM usage on a Cortex-M4 system

The following resource usage has been measured on a Cortex-M4 system using the SEGGER compiler with size optimization.

Addon	ROM
emNet SNMP Agent	approximately 6.2 kBytes

33.10.2 RAM usage

The following resource usage shows typical RAM requirements for an SNMPv2c Agent implementation. Most of the RAM is consumed for building the MIB tree in the application which is not subject of this measurement as it depends upon the application itself.

Addon	RAM
emNet SNMP Agent	approximately 300 Bytes

33.11 Resource usage (SNMPv3 USM)

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the SNMP Agent for SNMPv3 with User-based Security Model (USM) on top of an existing SNMPv2c configuration 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.

33.11.1 ROM usage on a Cortex-M4 system

The following resource usage has been measured on a Cortex-M4 system using the SEGGER compiler with size optimization.

Addon	ROM
emNet SNMP Agent SNMPv3 add-on with noAuthNoPriv	approximately 3.7 kBytes
emNet SNMP Agent SNMPv3 add-on with authPriv using AUTH(MD5) and PRIV(DES)	approximately 9.0 kBytes

33.11.2 RAM usage

Adding SNMPv3 support to an existing SNMPv2c application requires nearly zero additional RAM.

Chapter 34

CoAP client/server (Add-on)

The emNet Constrained Application Protocol (CoAP) client/server is an optional extension to emNet. The CoAP client/server can be used with emNet or with a different UDP/IP stack. All functions that are required to add a CoAP client/server to your application are described in this chapter.

34.1 emNet CoAP

The emNet CoAP client/server is an optional extension which adds CoAP support to the stack. It combines a maximum of performance with a small memory footprint. The CoAP server allows an embedded system to handle CoAP requests from a CoAP client. The CoAP client allows an embedded system to send request to a CoAP server.

The CoAP client/server implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 7252]	The Constrained Application Protocol (CoAP) Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc7252.txt*
[RFC 6690]	Constrained RESTful Environments (CoRE) Link Format Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc6690.txt*
[RFC 7641]	Observing Resources in the Constrained Application Protocol (CoAP) Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc7641.txt*
[RFC 7959]	Block-Wise Transfers in the Constrained Application Protocol (CoAP) Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc7959.txt*

The following table shows the contents of the emNet CoAP root directory:

Directory	Content
`.\Application\`	Contains the example application to run the CoAP client/server with emNet.
`.\Config\`	Contains the CoAP configuration file. Refer to *CoAP configuration* on page 1037 for detailed information.
`.\Inc\`	Contains the required include files.
`.\IP\`	Contains the CoAP sources and header files, `IP_COAP*`.
`.\Windows\IP\CoAP_Server\`	Contains the source, the project files and an executable to run the emNet CoAP server on a Microsoft Windows host. Refer to *Using the CoAP samples* on page 1034 for detailed information.
`.\Windows\IP\CoAP_Client\`	Contains the source, the project files and an executable to run the emNet CoAP client on a Microsoft Windows host. Refer to *Using the CoAP samples* on page 1034 for detailed information.

34.2 Feature list

- Low memory footprint.
- GET, DELETE, PUT, POST supported.
- Confirmable (CON) and non-confirmable (NON) requests supported.
- The server supports multiple clients.
- Independent of the UDP/IP stack: any stack with sockets can be used.
- Block transfer supported.
- Observe option supported.
- Example applications included.
- Demo with various option, request types, observable data included.
- Project for executable on PC for Microsoft Visual Studio included.

34.3 Requirements

UDP/IP stack

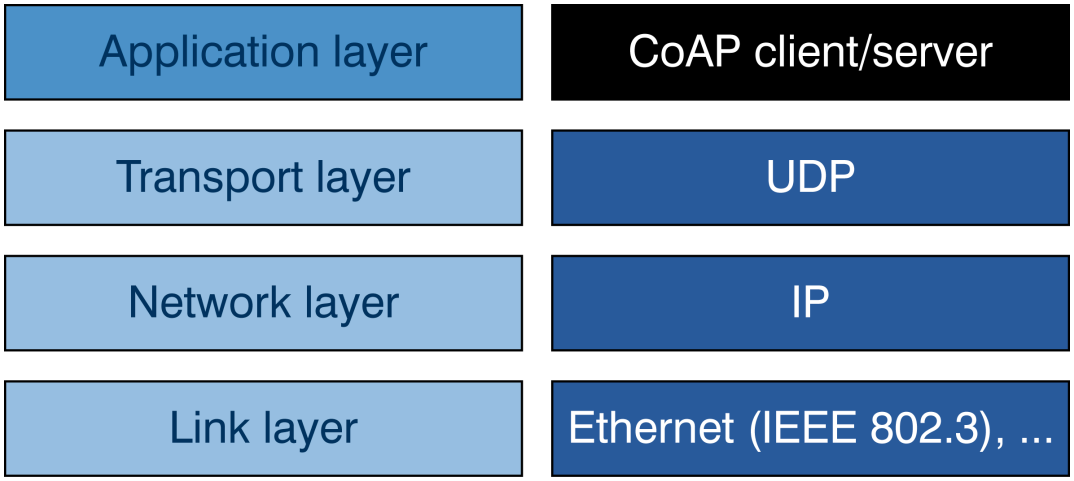
The emNet CoAP client/server requires an UDP/IP stack. It is optimized for emNet, but any RFC-compliant UDP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of emNet.

Multi tasking

The client/server doesn’t required any multi-tasking environment to run. But the various CoAP APIs are not thread safe and should be protected accordingly if used in different threads.

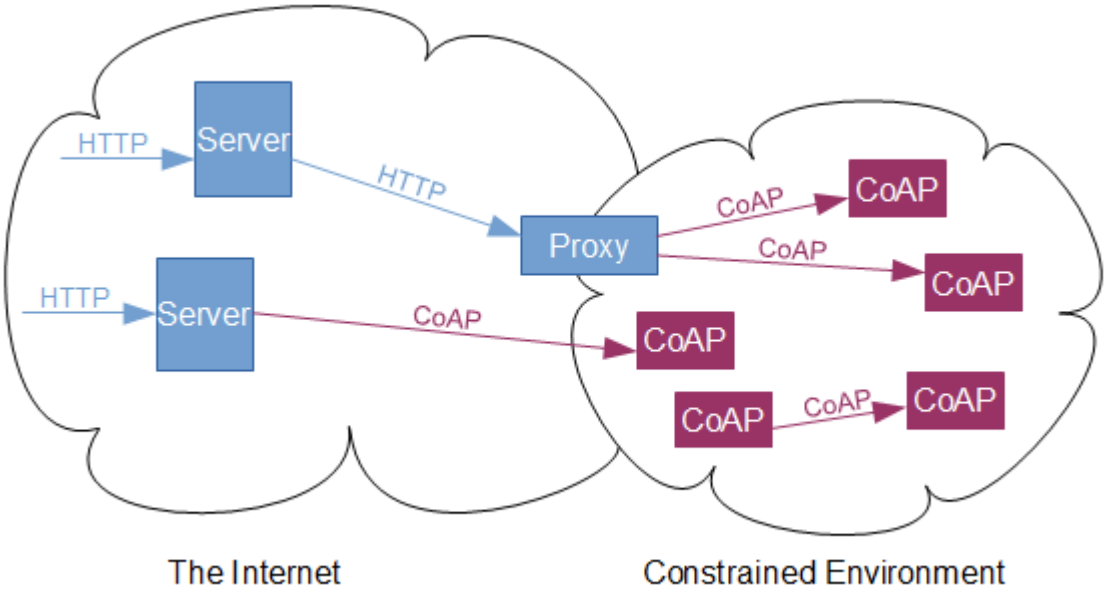
34.4 CoAP background

The Constrained Application Protocol (CoAP) is an Internet of Things (IoT) protocol which allows machine to machine (M2M) communication with small footprint applications. It uses Uniform Resource Identifier (URI) to identify resources on the server. With its small and simple 4 bytes header, it's ideal for devices with limited resources.



34.4.1 Protocol overview

The CoAP protocol implements a RESTful client-server based on four methods: GET, POST, PUT and DELETE. It could be embedded in the web transfer protocol thanks to CoAP proxys which could convert a web request (with address starting with coap://) into a CoAP request or used directly with a CoAP client.



The CoAP protocol is based on UDP (or DTLS for the secured version) and thus has to deal with potential message loss and repetition. It is a simple message exchange between endpoints. It relies on confirmable (CON) or non-confirmable (NON) messages.

Message type

They are four message types:

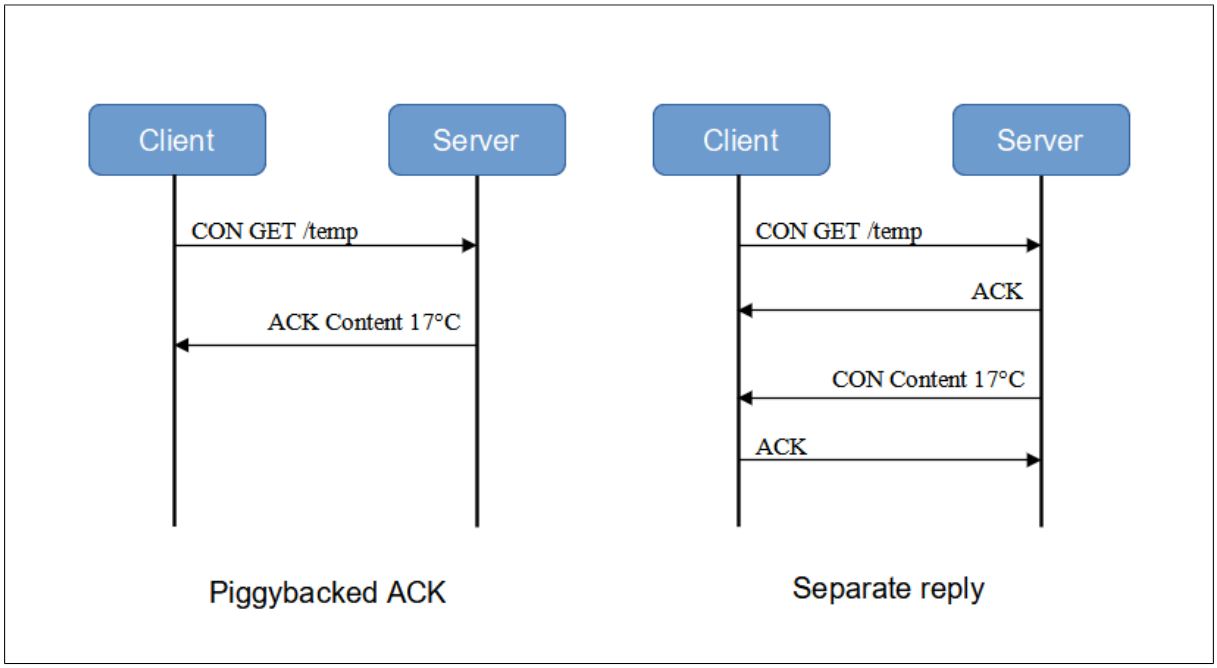
- **CON** messages used for requests. they expect to receive an acknowledgment (ACK) or a reset (RST), otherwise the message is retransmitted with an exponential back-off between retransmissions.
- **NON** messages don't expect any confirmation. They could be used as request or response.
- **ACK** messages are acknowledgment of a CON.
- **RST** messages are sent as a reply when the server is not able to process the request.

Requests

A request is sent with a message of type CON or NON. There are four possible request methods defined by the CoAP protocol:

- **GET**: The GET method retrieves a representation for the information that currently corresponds to the resource identified by the request URI.
- **PUT**: The PUT method requests that the resource identified by the request URI be updated with the enclosed representation.
- **POST**: The POST method requests that the representation enclosed in the request be processed to create a new resource or update an existing one.
- **DELETE**: The DELETE method requests that the resource identified by the request URI be deleted.

For example a client would like to receive the value of the URI "/temp" on a server. It could send a *"CON GET /temp"*. The server could directly answer with the data added to the ACK message (piggybacked ACK), or reply with an empty ACK and send a separate CON message with the data. This separate CON shall also be acknowledged.



Depending on the requests parameters, different reply codes or error codes could be expected.

34.4.2 Message format

CoAP messages are encoded in a simple binary format. The message format starts with a fixed-size 4-byte header. This is followed by a variable-length Token value, which can be between 0 and 8 bytes long.

Following the Token value comes a sequence of zero or more CoAP Options in Type-Length-Value (TLV) format, optionally followed by a payload that takes up the rest of the datagram.

The fields in the header are defined as follow:

Version (Ver):
2-bit unsigned integer. Indicates the CoAP version number. This implementation sets this field to 1 (01 binary).

Type (T):
2-bit unsigned integer. Indicates if this message is of type Confirmable (0), Non-confirmable (1), Acknowledgement (2), or Reset (3).

Token Length (TKL):
4-bit unsigned integer. Indicates the length of the variable-length Token field (0-8 bytes).

Code:
8-bit unsigned integer, split into a 3-bit class (most significant bits) and a 5-bit detail (least significant bits), documented as "c.dd" where "c" is a digit from 0 to 7 for the 3-bit subfield and "dd" are two digits from 00 to 31 for the 5-bit subfield. The class can indicate a request (0), a success response (2), a client error response (4), or a server error response (5). As a special case, Code 0.00 indicates an Empty message. In case of a request, the Code field indicates the Request Method; in case of a response, a Response Code.

Message ID:
16-bit unsigned integer in network byte order. Used to detect message duplication and to match messages of type Acknowledegment/Reset to messages of type Confirmable/Non-confirmable.

The header is followed by the Token value, which may be 0 to 8 bytes, as specified by the Token Length field. Header and Token are followed by zero or more Options. An Option can be followed by the end of the message, by another Option, or by the Payload Marker (`0xFF`) and the payload.

34.4.3 Response code

After receiving and interpreting a request, a server responds with a CoAP response. A response is identified by the Code field in the CoAP header being set to a Response Code. Similar to the HTTP Status Code, the CoAP Response Code indicates the result of the attempt to understand and satisfy the request.

```
 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|class|  detail |
+-+-+-+-+-+-+-+-+
```

The upper three bits of the 8-bit Response Code number define the class of response. The lower five bits do not have any categorization role; they give additional detail to the overall class

As a human-readable notation for specifications and protocol diagnostics, CoAP code numbers including the Response Code are documented in the format "c.dd", where "c" is the class in decimal, and "dd" is the detail as a two-digit decimal. For example, "Forbidden" is written as 4.03 -- indicating an 8-bit code value of hexadecimal `0x83` (4*`0x20`+3) or decimal 131 (4*32+3).

There are 3 classes of Response Codes:

- 2 - Success: The request was successfully received, understood, and accepted.
- 4 - Client Error: The request contains bad syntax or cannot be fulfilled.
- 5 - Server Error: The server failed to fulfill an apparently valid request.

Here are the basic response code from RFC 7252:

Code	Description
2.01	Created
2.02	Deleted
2.03	Valid
2.04	Changed
2.05	Content
4.00	Bad Request
4.01	Unauthorized
4.02	Bad Option
4.03	Forbidden
4.04	Not Found
4.05	Method Not Allowed
4.06	Not Acceptable
4.12	Precondition Failed
4.13	Request Entity Too Large
4.15	Unsupported Content-Format
5.00	Internal Server Error
5.01	Not Implemented
5.02	Bad Gateway
5.03	Service Unavailable
5.04	Gateway Timeout
5.05	Proxying Not Supported

34.4.4 CoAP options

Both requests and responses may include a list of one or more options. CoAP defines a single set of options that are used in both requests and responses:

Options are present to parametrize the request or the response. Some of the options are:

Content-Format:
Used to specify the format of a data (plain/text, binary, …).

ETag:
An entity-tag is intended for use as a resource-local identifier for differentiating between representations of the same resource that vary over time.
When sending a response to a request (typically a GET), the server may provide an ETag identifying the current state/value of the entity.
When a client sends a GET request with an ETag, the server can issue a 2.03 Valid response in place of a 2.05 Content response if the ETag match the current representation.

Max-Age:
The Max-Age option indicates the maximum time (in s) a response may be cached before it is considered not fresh

Uri-Path:
This is one of the option used to define the URI (with Uri-Host, Uri-Port and Uri-Query). Uri-Path specifies one segment of the absolute path to the resource (without the leading '/').

Accept:
The CoAP Accept option can be used to indicate which Content-Format is acceptable to the client.

If-Match:
The If-Match option may be used to make a request conditional on the current existence or value of an ETag for one or more representations of the target resource.
The value of an If-Match option is either an ETag or the empty string. An If-Match option with an ETag matches a representation with that exact ETag. The If-Match with an empty string matches all representations as long as the data exists already.
It is typically used in a PUT request to protect the update of a data.

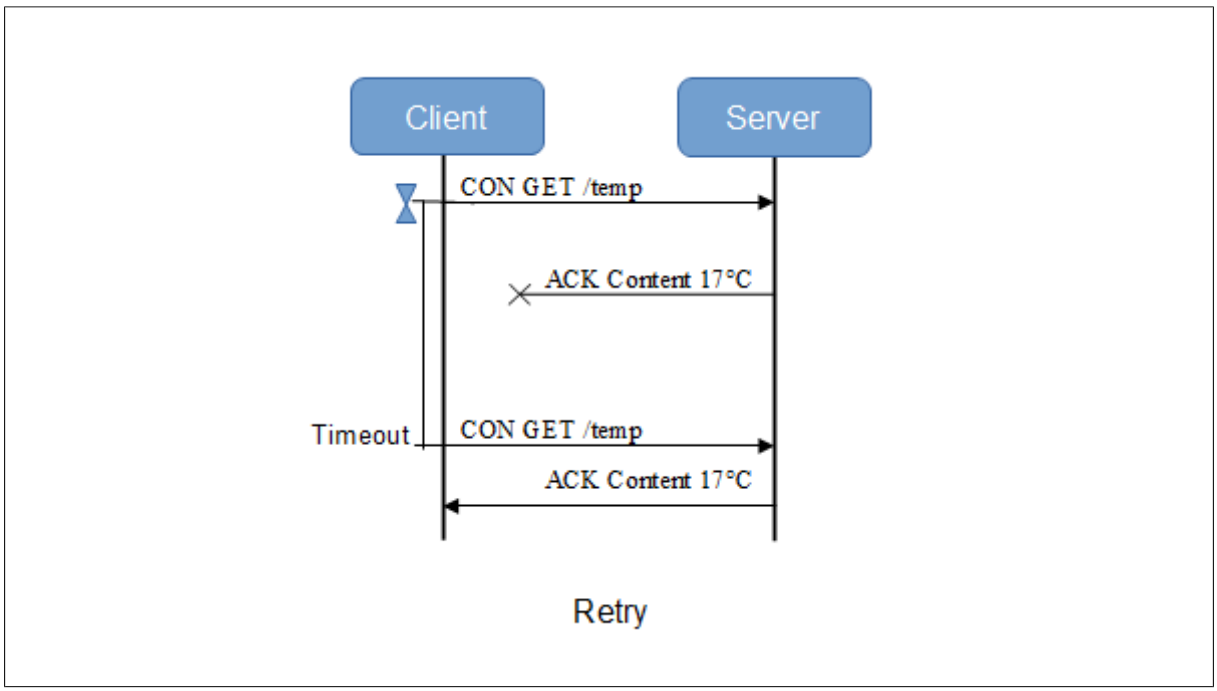
If-None-Match:
The If-None-Match option may be used to make a request conditional on the nonexistence of the target resource. If-None-Match is useful for resource creation requests, such as POST requests.

There are more options defined, refer to the RFC for the complete list.

Note that some options are repeatable. This means they could appear more than once per CoAP message.

34.4.5 Retry mechanism

Confirmable messages should be acknowledged. If the CON or the ACK doesn't reach its target, the CON is retransmitted with a exponential back-off between the retransmissions.



Per default, there are four retransmissions. The initial timeout between the initial message and the first retransmission is a random number between two and three seconds (with the default RFC values).

34.4.6 Block transfer

Basic CoAP messages work well for small payloads, however, it might be requested to transfer larger amount of data. CoAP is based on datagram transport layer and is thus subject to fragmentation.

Instead of relying on IP fragmentation, CoAP uses "Block" options in order to transfer multiple blocks of information from a resource representation in multiple request-response pairs.

Two block options are defined:

- Block1 is present in a request. Typically in a PUT or POST to send data to the server.
- Block2 is present in a response. Typically in a GET response to send the data in chunks.

The block option is composed of three values:

Block Size:
The possible block sizes are power of 2 values between 16 and 1024 included.

More Flag:
The more flag bit is set to 1 to indicate more blocks are to be expected. If set to 0 this means the current block is the last block of the transfer.

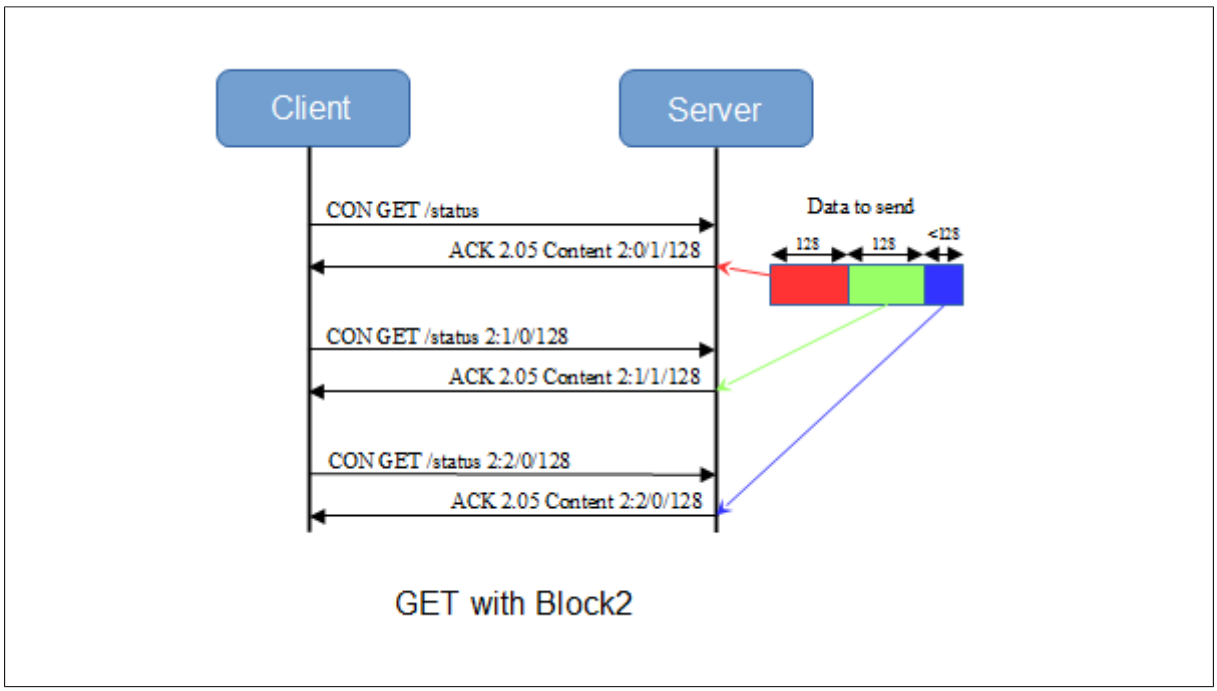
NUM:
This is the block number. Block number 0 indicates the first block.

The notation used to describe block is `<block type>:<NUM>/<More>/<Block Size>`. For example the first Block2 of a transfer where the More flag is set and the size of block is 128 bytes is written `2:0/1/128`.

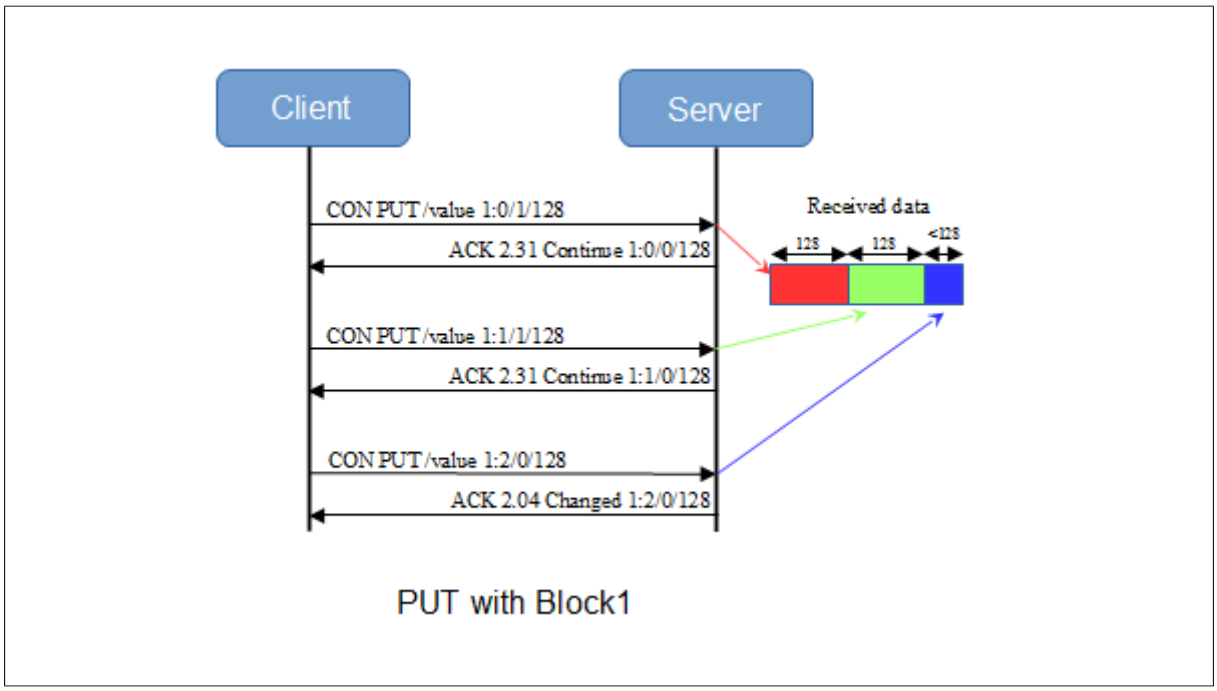
The block size used in the transfer is always the smaller of the sizes proposed by the client and the server. For example if a client makes a request with a block size of 128, but the server can only handle blocks of 64 bytes, then the server will reply with a block of 64 bytes and the following request from the client will also use blocks of 64 bytes.

There are two other options Size1 and Size2. Size1 (resp. Size2) is optionally used to give an indication of the total size of a Block1 (resp. Block2) transfer.

The following example gives a simple overview of a GET request with a Block2 option of 128 bytes.



Similarly with a PUT request with Block1 option.



34.4.7 Observe

The Observe option specifies a way for a client to register to a server resource. This allows a client to follow the evolution of a resource over a period of time.

The client (observer) registers to a server specific resource and receives a notification each time this resource is updated.

Request

In order to register, the client sends a GET with the Observe option set to 0. To de-register its interest, the client could either send another GET on the resource with Observe set to 1, or just “forget” the registered observe request. In this last case, when the server will send a notification on the resource, the client will reply with a RST message and the server will understand that the registration is not valid anymore.

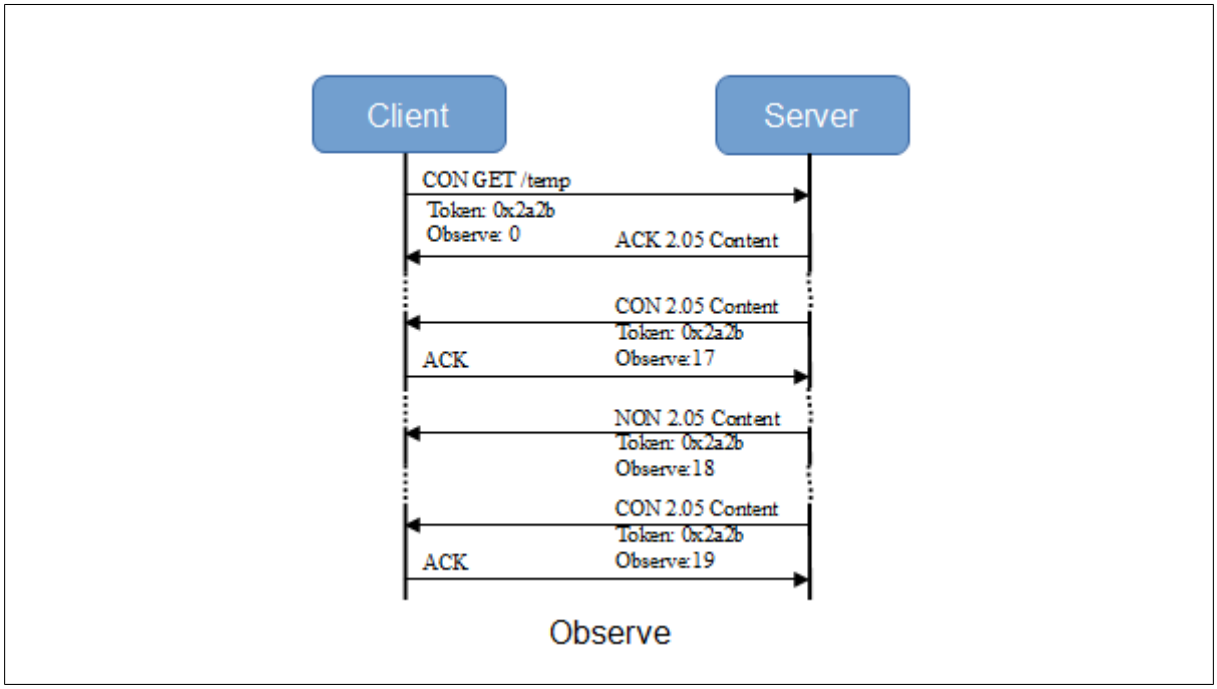
In order to identify the following notifications, the client must provide a token in the request.

Notification

Notifications are additional responses sent by the server in reply to the single extended GET request that created the registration. Each notification includes the token specified by the client in the request.

Notifications typically have a 2.05 (Content) response code. They include an Observe Option with a sequence number for reordering detection.

The notifications could have a different type than the one used for the request. For example if a registration is made with a CON, the server will reply with an ACK, but the following notifications could either be CON or NON or a mix of both. For example a server could decide to send a NON at a "MaxAge" period and to send a CON when the value is actually changing.



Of course, the observe notifications could use blocks when data to transfer is big. CON messages are also retransmitted if needed. If the retransmission maximum count is reached, the observation is canceled.

34.4.8 Built-In resource discovery

The discovery of resources offered by a CoAP endpoint is extremely important in machine-to-machine applications where there are no humans in the loop and static interfaces result in fragility.

Using the default UDP port, a client sends a GET request to the server with the Uri-Path set to ".well-known/core". The server will provide a response following the CoRE Link Format describing the resources present on the server.

An example of this discover reply is seen bellow. Note that line breaks were added for readability purpose and are not part of the actual message:

```
</obs>;obs;title="Simple observable data.",
</test>;title="test entry",
</separate>;title="simple variable with GET with LATE reply",
</BlockTransfer/ObsData>;obs;title="Observable data block.",
</ObsSensor>;obs;title="Observable value.",
</BlockTransfer/Data128bytes>;title="Textual value. GET and PUT, block support.
 Size 128 Bytes.",
</BlockTransfer/DelayedBlock>;title="Textual value. GET late reply with block
 support.",
</DataSupportingDelete>;title="Binary value (int). GET piggybacked, Support
 DELETE",
</temperature/average>;title="Textual value. GET late reply.",
```



```
</temperature/meas2>;title="Binary value (int). GET piggybacked",
</temperature/meas1>;ct=0;title="Textual or binary value. GET piggybacked, PUT",
</.well-known/core>
```

34.4.9 Implementation choices

A few choices were made for this CoAP implementation.

A POST on an existing resource is treated as a PUT. But a PUT with a URI that doesn't correspond to any resource is treated as an error (and not as a POST).

In order to reduce RAM usage, the implementation is stateless. This means in a block transfer it doesn't check the block sequence. It is indicating for example by clearing the "More" bit in a Block1 option when sending a "2.31 Continue" acknowledgment. It is of course possible for the application to perform this test as the block information is given in all payload related callbacks.

When the server receives a GET request on a branch (for example on "/temperature"), it will automatically reply with the list of all resources under this branch (like "/temperature/meas", "/temperature/average", …) unless a resource exists with the actual Uri-Path ("/temperature").

The server couldn't be used as a proxy. Therefore it doesn't support the Proxy-Uri and Proxy-Scheme options and replies with an error "5.05 Proxying Not Supported" in this case.

34.5 Using the CoAP samples

Ready to use examples for Microsoft Windows and emNet are supplied. If you use another UDP/IP stack, the samples have to be adapted.

34.5.1 Running the sample on target hardware

The emNet CoAP sample applications should always be the first step to check the proper function of the CoAP 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\`

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.

34.5.2 Using the Windows samples

If you have MS Visual C++ 6.00 or any later version available, you will be able to work with a Windows sample projects using the emNet CoAP client/server. If you do not have the Microsoft compiler, a precompiled executable is also supplied.

Building the sample program

Open the workspace .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.

34.5.3 Sample CoAP server application

The sample server application initializes and configures a CoAP server (`IP_COAP_SERVER_CONTEXT`) with a few resources (`IP_COAP_SERVER_DATA`). Each resource provides one or more handlers to process the requests (GET, PUT, …). These handlers are simple example and not representative of a full application.

The applications defines the number or possible simultaneous “connections” of the server. It’s not actual connections like with TCP, but more a context to handle a transfer. As a connection is closed once the transfer is completed, a server could handle way more clients than configured connections, especially when using confirmable messages since they are retransmitted when not acknowledged which could mitigate potential congestions.

After the server configurations, the server enters an infinite loop to call periodically the `IP_COAP_SERVER_Process()` function. This function does all the CoAP server processing:

- Handling of requests from clients.
- Retry mechanism of CON messages.
- Notification of observed resources.

34.5.4 Server callbacks description

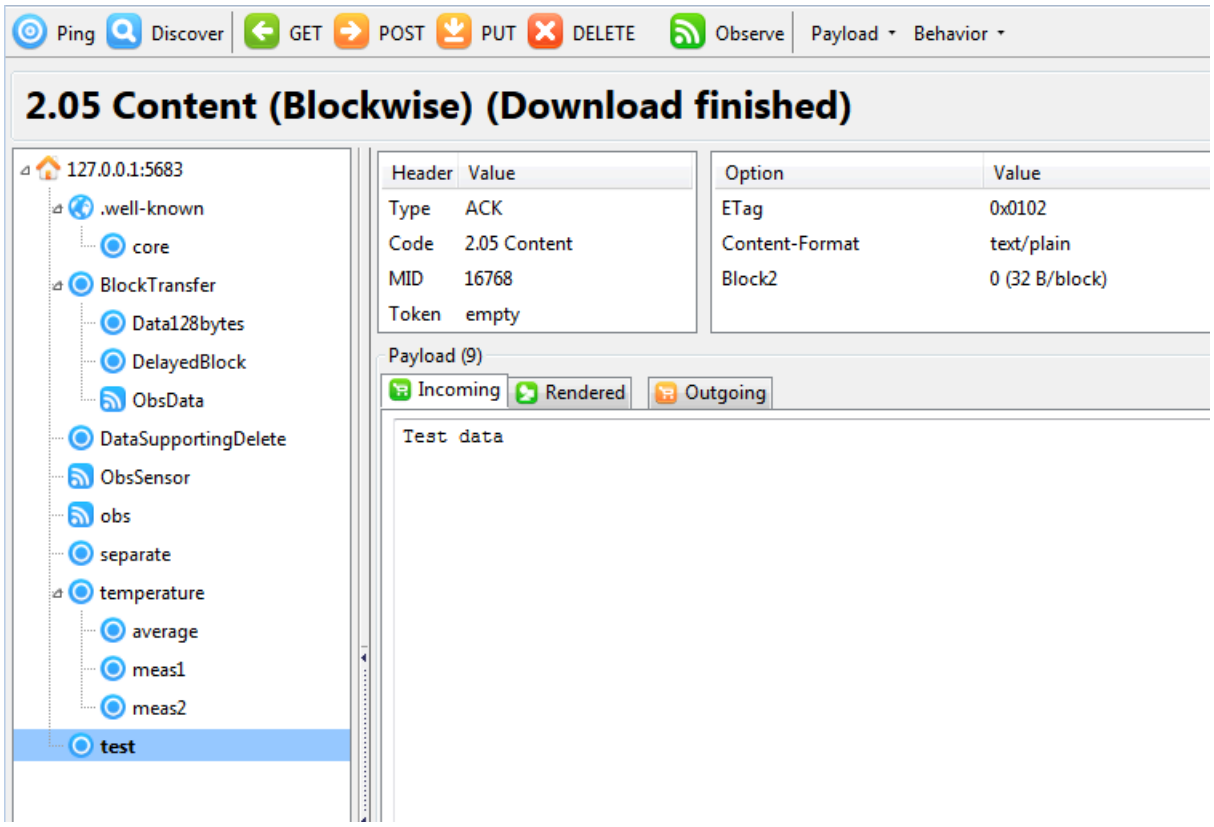
The resource are defined by the `IP_COAP_SERVER_DATA` and have several function pointers to allow and handle the various procedure.

All the callbacks have as parameter an `IP_COAP_CALLBACK_PARAM` which gives information on the message like header, options or block parameters.

Refer to *Callback pfGETPayload* on page 1116, *Callback pfPUTPayload* on page 1119 and *Callback pfDELHandler* on page 1121.

34.5.5 Testing the server

A possibility to test the server is to use the CoAP plug-in for firefox called Copper:



Start by sending a CoAP ping (`CON Empty`) to verify the server is receiving data. Then performs a discover (`GET .well-known/core`) to populate the resource tree.

34.5.6 Sample CoAP client application

The server address used for testing is defined in `COAP_SERVER`. By default it uses the test server coap://vs0.inf.ethz.ch. As the resource requested in the sample client application are also defined in the sample server application, the client could use the IP address of the running sample server application.

For example if both Windows samples are used, the `COAP_SERVER` define sould be set to the localhost 127.0.0.1.

The sample client application initializes first a CoAP client (`IP_COAP_CLIENT_CONTEXT`).

A client is composed of a requests array (`IP_COAP_CLIENT_REQUEST`). As a client may receive asynchronous messages (like observe notifications, retransmit of messages), it performs an infinite loop similarly to the server.

The main processing of the client could be summarized by:

```
while(1) {
  IP_COAP_CLIENT_Process(&_COAPClient);
  //
  // Check if there is a pending result.
  //
  r = IP_COAP_CLIENT_GetLastResult(&_COAPClient, &ResultCode, &pError, &ErrorLength);
  if (r >= 0) {
    _HandleResult(r, ResultCode, pError, ErrorLength);
```



```
    }
    //
    // Check if there is something to do.
    //
    if (SomethingToDo) {
      //
      // Check if there is a free request available.
      //
      if (IP_COAP_CLIENT_GetFreeRequestIdx(&_COAPClient, &Index) == IP_COAP_RETURN_OK) {
        //
        // Initialize the new request.
        //
        r = IP_COAP_CLIENT_SetCommand(&_COAPClient, Index, ...);
        r = IP_COAP_CLIENT_SetOptionXxx(&_COAPClient, Index, ...);
        //
        // Send the request.
        //
        r = IP_COAP_CLIENT_BuildAndSend(&_COAPClient, Index);
      }
    }
  };
```

This way the client could handle requests and observations in parallel.

If the client is using only one request (i.e. connection) at a time (for example NSTART is defined as 1 as per default), and if the client doesn't make use of the observe functionality, the main loop could be avoided and instead have just a simpler request handler.

The processing would be similar to:

```
  //
  // Check if there is a free request to execute a new one.
  //
  if (IP_COAP_CLIENT_GetFreeRequestIdx(&_COAPClient, &Index) == IP_COAP_RETURN_OK) {
    //
    // Initialize the new request.
    //
    r = IP_COAP_CLIENT_SetCommand(&_COAPClient, Index, ...);
    r = IP_COAP_CLIENT_SetOptionXxx(&_COAPClient, Index, ...);
    //
    // Send the request.
    //
    r = IP_COAP_CLIENT_BuildAndSend(&_COAPClient, Index);
    //
    // Do CoAP processing until the reply is received.
    //
    do {
      IP_COAP_CLIENT_Process(&_COAPClient);
      //
      // Get last result.
      //
      r = IP_COAP_CLIENT_GetLastResult(&_COAPClient, &ResultCode, &pError, &ErrorLength);
    } while (r < 0);
    _HandleResult(r, ResultCode, pError, ErrorLength);
  }
```

34.5.7 Client callbacks description

When sending a GET, POST or PUT request, the user is invited to define a callback to handle the payload through the function `IP_COAP_CLIENT_SetPayloadHandler()`.

A payload callback have as parameter an `IP_COAP_CALLBACK_PARAM` which gives information on the message like header, options or block parameters.

Refer to *Callback PF_CLIENT_PAYLOAD* on page 1125.

34.6 CoAP configuration

The emNet CoAP client/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.

34.6.1 CoAP 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).

34.6.2 Configuration switches

Type	Symbolic name	Default	Description
F	`IP_COAP_WARN`	--	Defines a function to output warnings. In debug configurations (`DEBUG = 1`) `IP_COAP_WARN` maps to `IP_Warnf_Application()`.
F	`IP_COAP_LOG`	--	Defines a function to output logging messages. In debug configurations (`DEBUG = 1`) `IP_COAP_LOG` maps to `IP_Logf_Application()`.
N	`IP_COAP_NSTART`	1	Number of connections possible at the same time. This value shall match the CoAP network setting.
N	`IP_COAP_MAX_RETRANSMIT`	4	Maximum number of retransmission of a CON message. This value shall match the CoAP network setting.
N	`IP_COAP_ACK_TIMEOUT`	2000	Time [ms] to wait for an ACK. Please note that this is the base timeout to wait. This value gets randomized between 1x and 1.5x for the first CON and is used as is for the last retry. For all other retries the randomized value gets doubled for each retry.

Type	Symbolic name	Default	Description
N	IP_COAP_DEFAULT_LEISURE	5000	Default leisure time in milliseconds to randomize the time to reply to a multicast NON request.
N	IP_COAP_OBS_FORCE_CON_TIMEOUT	2000	Time [ms] after which the next observable report is sent as CON instead of NON to check if the client is still alive. The RFC suggests maximum 24 hours, we use 12 by default.
F	IP_COAP_MEMSET	memset	memset function.
F	IP_COAP_MEMCPY	memcpy	memcpy function.
F	IP_COAP_MEMCMP	memcmp	memcmp function.
F	IP_COAP_MEMMOVE	memmove	memmove function.
F	IP_COAP_STRLEN	strlen	strlen function.

34.7 API functions

Function	Description
Server	
IP_COAP_SERVER_Init()	Initializes the CoAP server context.
IP_COAP_SERVER_Process()	Called periodically to handle CoAP server processing: handles pending actions, read and handle received packet.
IP_COAP_SERVER_GetMsgBuffer()	Returns the buffer used to send messages.
IP_COAP_SERVER_AddData()	Adds a data entry to the server.
IP_COAP_SERVER_RemoveData()	Removes a previously configured data from the server.
IP_COAP_SERVER_AddClientBuffer()	Adds a pool of client structures to be used by the server to handle request.
IP_COAP_SERVER_AddObserverBuffer()	Adds a pool of observer structures to be used by the server to handle observer request.
IP_COAP_SERVER_UpdateData()	Called by the user to indicate data are ready or changed.
IP_COAP_SERVER_SetDefaultBlockSize()	Configures the default block size used by the server.
IP_COAP_SERVER_SetPOSTHandler()	Configures the callback used when receiving a POST request to create a new server data entry.
IP_COAP_SERVER_ConfigSet()	Activates the given options from the server configuration.
IP_COAP_SERVER_ConfigClear()	Clears the given options from the server configuration.
IP_COAP_SERVER_SetURIPort()	Configures the UDP port to be sent as URI-Port in reply to a GET.
IP_COAP_SERVER_SetHostName()	Configures the host name to be sent as URI-Host in reply to a GET.
IP_COAP_SERVER_SetErrorDescription()	Configures a temporary error description to be sent as payload in an error message.
Client	
IP_COAP_CLIENT_Init()	Initializes a CoAP client.
IP_COAP_CLIENT_Process()	Periodic function called to manage CoAP request and observer of a client.
IP_COAP_CLIENT_GetFreeRequestIdx()	Returns a free request.
IP_COAP_CLIENT_AbortRequestIdx()	Aborts an on-going request.
IP_COAP_CLIENT_SetServerAddress()	Configures the UDP parameters of the server.
IP_COAP_CLIENT_SetDefaultBlockSize()	Sets the default block size to be used by the client.
IP_COAP_CLIENT_SetCommand()	Configures the request type and code.
IP_COAP_CLIENT_SetToken()	Configures the token of a request.
IP_COAP_CLIENT_SetPayloadHandler()	Configures the callback used to manage the payload of the request.
IP_COAP_CLIENT_SetReplyWaitTime()	Configures the time to wait for the reply after sending the request.

Function	Description
IP_COAP_CLIENT_BuildAndSend()	Builds the CoAP message and sends it based on a configured request.
IP_COAP_CLIENT_GetLastResult()	Returns the status and information when a request is completed.
IP_COAP_CLIENT_GetMsgBuffer()	Returns the buffer used to send messages.
IP_COAP_CLIENT_GetLocationPath()	Returns the Location-Path received.
IP_COAP_CLIENT_GetLocationQuery()	Returns the Location-Query received.
IP_COAP_CLIENT_SetOptionURIPath()	Adds the URI-Path option to a request.
IP_COAP_CLIENT_SetOptionURIHost()	Adds the URI-Host option to a request.
IP_COAP_CLIENT_SetOptionURIPort()	Adds the URI-Port option to a request.
IP_COAP_CLIENT_SetOptionURIQuery()	Adds the URI-Query option to a request.
IP_COAP_CLIENT_SetOptionETag()	Adds the ETag option to a request.
IP_COAP_CLIENT_SetOptionBlock()	Adds the block option to a request.
IP_COAP_CLIENT_SetOptionAccept()	Adds the Accept option to a request.
IP_COAP_CLIENT_SetOptionContentFormat()	Adds the Content-Format option to a request.
IP_COAP_CLIENT_SetOptionIfNoneMatch()	Adds the If-None-Match option to a request.
IP_COAP_CLIENT_SetOptionLocationPath()	Adds the location path option to a request.
IP_COAP_CLIENT_SetOptionLocationQuery()	Adds the location query option to a request.
IP_COAP_CLIENT_SetOptionProxyURI()	Adds the proxy URI option to a request.
IP_COAP_CLIENT_SetOptionProxyScheme()	Adds the proxy scheme option to a request.
IP_COAP_CLIENT_SetOptionSize1()	Adds the Size1 option to a request.
IP_COAP_CLIENT_SetOptionAddIFMatch()	Adds the If-Match option to a request.
IP_COAP_CLIENT_OBS_Init()	Initializes an observe request.
IP_COAP_CLIENT_OBS_Abort()	Stops an active observer.
IP_COAP_CLIENT_OBS_SetEndCallback()	Configures the end callback for an observe request.
Utility	
IP_COAP_CheckAcceptFormat()	Verifies if the Accept-Format matches the requested one.
IP_COAP_GetAcceptFormat()	Reads the Accept-Format option value.
IP_COAP_CheckContentFormat()	Verifies if the Content-Format match the requested one.
IP_COAP_GetContentFormat()	Reads the Content-Format option value.
IP_COAP_IsLastBlock()	Checks if the current block is the last one of the transfer.
IP_COAP_GetURIHost()	Sets a pointer on the start of the URI-HostName.
IP_COAP_GetURIPath()	Sets a pointer on the start of the URI-Path.
IP_COAP_GetURIPort()	Reads the URI-Port option value.
IP_COAP_GetQuery()	Sets a pointer on the start of the Query field.
IP_COAP_GetETag()	Sets a pointer on the ETag value.

Function	Description
IP_COAP_GetMaxAge()	Reads the MaxAge option value.
IP_COAP_GetSize1()	Reads the Size1 option value.
IP_COAP_GetSize2()	Reads the Size2 option value.
IP_COAP_GetLocationPath()	Sets a pointer on the start of the Location-Path.
IP_COAP_GetLocationQuery()	Sets a pointer on the start of the Location-Query field.

34.7.1 Server

34.7.1.1 IP_COAP_SERVER_Init()

Description

Initializes the CoAP server context. This function needs to be called first.

Prototype

```
int IP_COAP_SERVER_Init(      IP_COAP_SERVER_CONTEXT * pContext,
                              U8                     * pMsgBuffer,
                              U16                      MsgBufferSize,
                        const IP_COAP_API            * pAPI);
```

Parameters

Parameter	Description
pContext	Server context to initialize.
pMsgBuffer	Buffer used to handle UDP packets.
MsgBufferSize	Size of pMsgBuffer in bytes.
pAPI	API used for UDP transfer and time.

Return value

= IP_COAP_RETURN_OK Success.
≠ IP_COAP_RETURN_OK Error.

Additional information

If possible, the message buffer should be as big as the MTU but since fragmentation is not used, this is useless to have a message buffer bigger than 1500 bytes.

The default block size will be adjusted to take into account the buffer size. Thus if the buffer is small (i.e. 512 bytes) the default block size will be adjusted (to 256 in this case).

APIs are used to send and receive UDP packets. The third one is used to retrieve the time in ms.

Example

```
static IP_COAP_SERVER_CONTEXT      _COAPServer;
static IP_COAP_CONN_INFO           _ConnInfo;
static U8                          _MsgBuffer[1500];
static IP_COAP_SERVER_CLIENT_INFO  _COAPClientInfo[IP_COAP_MAX_NUM_CLIENT];
static IP_COAP_OBSERVER            _COAPObservers[IP_COAP_MAX_NUM_OBS];

static const IP_COAP_API _APP_Api = {
  _APP_Receive,
  _APP_Send,
  _APP_GetTimeMs
};

void Server(void) {
  //
  // Initializes the CoAP server.
  //
  IP_COAP_SERVER_Init(&_COAPServer, _MsgBuffer, sizeof(_MsgBuffer), &_APP_Api);
  //
  // Add the clients and observers buffers.
  //
  IP_COAP_SERVER_AddClientBuffer(&_COAPServer, &_COAPClientInfo[0], IP_COAP_MAX_NUM_CLIENT);
  IP_COAP_SERVER_AddObserverBuffer(&_COAPServer, &_COAPObservers[0], IP_COAP_MAX_NUM_OBS);
  //
  // Add a handler to perform POST command.
  //
  IP_COAP_SERVER_SetPOSTHandler(&_COAPServer, _POSTCreateEntry);
  //
  // Configure a default block size.
  //
```



```
  IP_COAP_SERVER_SetDefaultBlockSize(&_COAPServer, 32);
  //
  // Configuration of the server by adding ServerData structures.
  //
  _ConfigureServer();
  //
  _ConnInfo.Family   = IP_COAP_IPV4;
  _ConnInfo.Port     = IP_COAP_DEFAULT_PORT;
  //
  // Run the main loop.
  //
  while (1) {
    //
    // Call regularly the CoAP processing function.
    //
    IP_COAP_SERVER_Process(&_COAPServer, &_ConnInfo);
    //
    // Perform the application processing.
    //
    _ApplicationProcessing();
  }
}
```


34.7.1.2 IP_COAP_SERVER_Process()

Description

Called periodically to handle CoAP server processing: handles pending actions, read and handle received packet.

Prototype

```
int IP_COAP_SERVER_Process(IP_COAP_SERVER_CONTEXT * pContext,
                           IP_COAP_CONN_INFO      * pConnInfo);
```

Parameters

Parameter	Description
pContext	Pointer to the server CoAP context.
pConnInfo	Pointer to the connection info used to receive UDP packets.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

IP_COAP_SERVER_Process() is not thread safe. If it is used in a multithreading environment, it shall be proofed by the user that no calls to configuration functions (such as _AddData(), _RemoveData(), …) are made while this function runs.

Refer to *IP_COAP_SERVER_Init* on page 1042 for usage example.

34.7.1.3 IP_COAP_SERVER_GetMsgBuffer()

Description

Returns the buffer used to send messages.

Prototype

```
U8 *IP_COAP_SERVER_GetMsgBuffer(IP_COAP_SERVER_CONTEXT * pContext,
                                U16                    * pMsgLength);
```

Parameters

Parameter	Description
pContext	Server context.
pMsgLength	Filled with buffer length. Can be NULL.

Return value

≠ NULL Pointer to the message buffer.

34.7.1.4 IP_COAP_SERVER_AddData()

Description

Adds a data entry to the server.

Prototype

```
int IP_COAP_SERVER_AddData(IP_COAP_SERVER_CONTEXT * pContext,
                           IP_COAP_SERVER_DATA    * pData);
```

Parameters

Parameter	Description
pContext	Server context.
pData	Data to add.

Return value

= IP_COAP_RETURN_OK Success.
≠ IP_COAP_RETURN_OK Error.

Refer to *Structure IP_COAP_SERVER_DATA* on page 1111 for usage example.

34.7.1.5 IP_COAP_SERVER_RemoveData()

Description

Removes a previously configured data from the server.

Prototype

```
int IP_COAP_SERVER_RemoveData(IP_COAP_SERVER_CONTEXT * pContext,
                              IP_COAP_SERVER_DATA    * pData);
```

Parameters

Parameter	Description
pContext	Server context.
pData	Data to remove.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error, data not found.

34.7.1.6 IP_COAP_SERVER_AddClientBuffer()

Description

Adds a pool of client structures to be used by the server to handle request. The number of clients represents the number of clients that could be handled at the same time (request on-going).

Prototype

```
int IP_COAP_SERVER_AddClientBuffer(IP_COAP_SERVER_CONTEXT     * pContext,
                                   IP_COAP_SERVER_CLIENT_INFO * pClientInfo,
                                   unsigned                     NumClientInfo);
```

Parameters

Parameter	Description
pContext	Server context.
pClientInfo	Start of the client pool.
NumClientInfo	Number of clients in the given pool.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

At least one client is mandatory. Without any client configured the server won't be able to handle requests.

This represents the total number of clients/connections active at the same time. Since a connection is closed when the transfer is completed, a server could handle much more clients than this configuration, depending on the traffic generated by each client.

Refer to *IP_COAP_SERVER_Init* on page 1042 for usage example.

34.7.1.7 IP_COAP_SERVER_AddObserverBuffer()

Description

Adds a pool of observer structures to be used by the server to handle observer request. When the pool is full, no more observer request could be handled.

Prototype

```
int IP_COAP_SERVER_AddObserverBuffer(IP_COAP_SERVER_CONTEXT * pContext,
                                     IP_COAP_OBSERVER       * pObs,
                                     unsigned                 NumObservers);
```

Parameters

Parameter	Description
pContext	Server context.
pObs	Start of the observer pool.
NumObservers	Number of observers in the given pool.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Refer to *IP_COAP_SERVER_Init* on page 1042 for usage example.

34.7.1.8 IP_COAP_SERVER_UpdateData()

Description

Called by the user to indicate data are ready or changed. It is used in two cases:

- GET with a separate reply (ACK then reply in another request) to indicate the reply is ready to be sent.
- Observable data has changed. This needs to be called each time an observable data changes.

Prototype

```
int IP_COAP_SERVER_UpdateData(IP_COAP_SERVER_CONTEXT * pContext,
                              IP_COAP_SERVER_DATA    * pData,
                              U8                       ObsUpdateType,
                              unsigned                 AutoETag);
```

Parameters

Parameter	Description
pContext	Server context.
pData	Data that is ready or changed
ObsUpdateType	Type of the request to send in case of observable data update: • IP_COAP_TYPE_CON. • IP_COAP_TYPE_NON.
AutoETag	Flag to generate a new ETag automatically. • 1: Automatic ETag increment. • 0: No change on ETag.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

When set to 1, AutoETag flag will increase by 1 the ETag value.

34.7.1.9 IP_COAP_SERVER_SetDefaultBlockSize()

Description

Configures the default block size used by the server.

Prototype

```
int IP_COAP_SERVER_SetDefaultBlockSize(IP_COAP_SERVER_CONTEXT * pContext,
                                       U16                      BlockSize);
```

Parameters

Parameter	Description
pContext	Server context.
BlockSize	Requested block size in bytes.

Return value

= IP_COAP_RETURN_OK Success.
≠ IP_COAP_RETURN_OK Error.

Refer to *IP_COAP_SERVER_Init* on page 1042 for usage example.

34.7.1.10 IP_COAP_SERVER_SetPOSTHandler()

Description

Configures the callback used when receiving a POST request to create a new server data entry.

Prototype

```
int IP_COAP_SERVER_SetPOSTHandler(IP_COAP_SERVER_CONTEXT * pContext,
                                  PF_POST_HANDLER          pfPOSTCreateEntry);
```

Parameters

Parameter	Description
pContext	Server context.
pfPOSTCreateEntry	Pointer to the function to call.

Return value

= IP_COAP_RETURN_OK Success.
≠ IP_COAP_RETURN_OK Error.

Refer to *IP_COAP_SERVER_Init* on page 1042 for usage example.

34.7.1.11 IP_COAP_SERVER_ConfigSet()

Description

Activates the given options from the server configuration.

Prototype

```
int IP_COAP_SERVER_ConfigSet(IP_COAP_SERVER_CONTEXT * pContext,
                             U8                       ConfigMask);
```

Parameters

Parameter	Description
pContext	Server context.
ConfigMask	Mask of the configurable options (IP_COAP_CONFIG_xxx).

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

The possible options to configure are:

- IP_COAP_CONFIG_DISABLE_BLOCK1 to disable Block1 usage.
- IP_COAP_CONFIG_DISABLE_BLOCK2 to disable Block2 usage.
- IP_COAP_CONFIG_DISABLE_BLOCKS to disable both Block1 and Block2.
- IP_COAP_CONFIG_DISABLE_OBSERVE to disable Observe option support.

34.7.1.12 IP_COAP_SERVER_ConfigClear()

Description

Clears the given options from the server configuration.

Prototype

```
int IP_COAP_SERVER_ConfigClear(IP_COAP_SERVER_CONTEXT * pContext,
                               U8                       ConfigMask);
```

Parameters

Parameter	Description
pContext	Server context.
ConfigMask	Mask of the configurable options (IP_COAP_CONFIG_xxx).

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

The possible options to configure are:

- IP_COAP_CONFIG_DISABLE_BLOCK1 to re-enable Block1 usage.
- IP_COAP_CONFIG_DISABLE_BLOCK2 to re-enable Block2 usage.
- IP_COAP_CONFIG_DISABLE_BLOCKS to re-enable both Block1 and Block2.
- IP_COAP_CONFIG_DISABLE_OBSERVE to re-enable Observe option support.

34.7.1.13 IP_COAP_SERVER_SetURIPort()

Description

Configures the UDP port to be sent as URI-`Port` in reply to a GET.

Prototype

```
int IP_COAP_SERVER_SetURIPort(IP_COAP_SERVER_CONTEXT * pContext,
                              U16                      Port);
```

Parameters

Parameter	Description
`pContext`	Server context.
`Port`	UDP port.

Return value

= `IP_COAP_RETURN_OK`	Success.
≠ `IP_COAP_RETURN_OK`	Error.

Additional information

The server data should be configured to send the URI-`Port` option in a GET reply.

It has no impact on the actual port used in the `IP_COAP_CONN_INFO` structure.

34.7.1.14 IP_COAP_SERVER_SetHostName()

Description

Configures the host name to be sent as URI-Host in reply to a GET.

Prototype

```
int IP_COAP_SERVER_SetHostName(      IP_COAP_SERVER_CONTEXT * pContext,
                               const char                   * sHostName);
```

Parameters

Parameter	Description
pContext	Server context.
sHostName	Null terminating string with the hostname.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

The server data should be configured to send the URI-Host option in a GET reply.

34.7.1.15 IP_COAP_SERVER_SetErrorDescription()

Description

Configures a temporary error description to be sent as payload in an error message. This is typically called from payload callbacks.

Prototype

```
int IP_COAP_SERVER_SetErrorDescription(      IP_COAP_SERVER_CONTEXT * pContext,
                                       const char                   * sErrorDesc);
```

Parameters

Parameter	Description
pContext	Server context.
sErrorDesc	Null terminating string with the error description.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

34.7.2 Client

34.7.2.1 IP_COAP_CLIENT_Init()

Description

Initializes a CoAP client.

Prototype

```
int IP_COAP_CLIENT_Init(      IP_COAP_CLIENT_CONTEXT * pContext,
                              U8                     * pMsgBuffer,
                              U16                      MsgBufferSize,
                        const IP_COAP_API            * pAPI);
```

Parameters

Parameter	Description
pContext	Client context.
pMsgBuffer	Buffer used to handle UDP packets.
MsgBufferSize	Size of pMsgBuffer in bytes.
pAPI	API used for UDP transfer and time.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

If possible, the message buffer should be as big as the MTU but since fragmentation is not used, this is useless to have a message buffer bigger than 1500 bytes.

The default block size will be adjusted to take into account the buffer size. Thus if the buffer is small (i.e. 512 bytes) the default block size will be adjusted (to 256 in this case).

APIs are used to send and receive UDP packets. The third one is used to retrieve the time in ms.

Example

```
static IP_COAP_CLIENT_CONTEXT  _COAPClient;
static IP_COAP_CONN_INFO       _ConnInfo;
static U8                      _MsgBuffer[1500];

static const IP_COAP_API _APP_Api = {
  _APP_Receive,
  _APP_Send,
  _APP_GetTimeMs
};

void _ClientInit(void) {
  //
  // Initializes the CoAP client.
  //
  IP_COAP_CLIENT_Init(&_COAPClient, &_MsgBuffer[0], sizeof(_MsgBuffer), &_APP_Api);
  //
  // Configure server address.
  //
  _ConnInfo.Family   = IP_COAP_IPV4;
  _ConnInfo.Port     = COAP_PORT;
  _ConnInfo.IPAddrV4 = COAP_SERVER_ADDRESS;
  IP_COAP_CLIENT_SetServerAddress(&_COAPClient, &_ConnInfo);
}
```


34.7.2.2 IP_COAP_CLIENT_Process()

Description

Periodic function called to manage CoAP request and observer of a client.

Prototype

```
int IP_COAP_CLIENT_Process(IP_COAP_CLIENT_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	Client context.

Return value

= IP_COAP_RETURN_OK	A free request is present.
≠ IP_COAP_RETURN_OK	No free request available.

Refer to *Sample CoAP client application* on page 1035 for usage example.

34.7.2.3 IP_COAP_CLIENT_GetFreeRequestIdx()

Description

Returns a free request. Should be called to get the index of a free request to be used to configure and send the request.

Prototype

```
int IP_COAP_CLIENT_GetFreeRequestIdx(IP_COAP_CLIENT_CONTEXT * pContext,
                                     unsigned              * pIndex);
```

Parameters

Parameter	Description
pContext	Client context.
pIndex	Filled with the free request index.

Return value

= IP_COAP_RETURN_OK	A free request is present.
≠ IP_COAP_RETURN_OK	No free request available.

Refer to *Sample CoAP client application* on page 1035 for usage example.

34.7.2.4 IP_COAP_CLIENT_AbortRequestIdx()

Description

Aborts an on-going request.

Prototype

```
int IP_COAP_CLIENT_AbortRequestIdx(IP_COAP_CLIENT_CONTEXT * pContext,
                                   unsigned                 Index);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the request to abort.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

34.7.2.5 IP_COAP_CLIENT_SetServerAddress()

Description

Configures the UDP parameters of the server.

Prototype

```
int IP_COAP_CLIENT_SetServerAddress(IP_COAP_CLIENT_CONTEXT * pContext,
                                    IP_COAP_CONN_INFO      * pConnInfo);
```

Parameters

Parameter	Description
pContext	Client context.
pConnInfo	UDP connection information.

Return value

= IP_COAP_RETURN_OK Success.
≠ IP_COAP_RETURN_OK Error.

Refer to *IP_COAP_CLIENT_Init* on page 1059 for usage example.

34.7.2.6 IP_COAP_CLIENT_SetDefaultBlockSize()

Description

Sets the default block size to be used by the client.

Prototype

```
int IP_COAP_CLIENT_SetDefaultBlockSize(IP_COAP_CLIENT_CONTEXT * pContext,
                                       U16                      BlockSize);
```

Parameters

Parameter	Description
pContext	Client context.
BlockSize	Size of the block in bytes.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

The given block size will be adjusted if it is too big for the configured message buffer size.

A specific request could still be configured with another block size by calling IP_COAP_CLIENT_SetOptionBlock().

34.7.2.7 IP_COAP_CLIENT_SetCommand()

Description

Configures the request type and code.

Prototype

```
int IP_COAP_CLIENT_SetCommand(IP_COAP_CLIENT_CONTEXT * pContext,
                              unsigned                 Index,
                              U8                       Type,
                              U8                       Code);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Free request index.
Type	Type of the request: • IP_COAP_TYPE_CON. • IP_COAP_TYPE_NON.
Code	Code of the request: • IP_COAP_CODE_REQ_GET. • IP_COAP_CODE_REQ_DEL. • IP_COAP_CODE_REQ_PUT. • IP_COAP_CODE_REQ_POST.

Return value

= IP_COAP_RETURN_OK Success.
≠ IP_COAP_RETURN_OK Error.

Example

```
//
// Build the request with the given free index.
//
IP_COAP_CLIENT_SetCommand(&_COAPClient, Index, IP_COAP_TYPE_CON, IP_COAP_CODE_REQ_GET);
IP_COAP_CLIENT_SetOptionURIPath(&_COAPClient, Index, (U8*)".well-known/core", 16);
IP_COAP_CLIENT_SetOptionBlock(&_COAPClient, Index, 128);
IP_COAP_CLIENT_SetPayloadHandler(&_COAPClient, Index, _DISCOVER_Handler);
//
r = IP_COAP_CLIENT_BuildAndSend(&_COAPClient, Index);
```


34.7.2.8 IP_COAP_CLIENT_SetToken()

Description

Configures the token of a request.

Prototype

```
int IP_COAP_CLIENT_SetToken(IP_COAP_CLIENT_CONTEXT * pContext,
                            unsigned                 Index,
                            U8                     * pToken,
                            U8                       TokenLength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Free request index.
pToken	Pointer to the token.
TokenLength	Length of the token.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

34.7.2.9 IP_COAP_CLIENT_SetPayloadHandler()

Description

Configures the callback used to manage the payload of the request. For GET it's the function that will be called with the data when receiving the reply. For PUT/POST, this is the function that will be used to add the data in the request message.

Prototype

```
int IP_COAP_CLIENT_SetPayloadHandler(IP_COAP_CLIENT_CONTEXT * pContext,
                                     unsigned                 Index,
                                     PF_CLIENT_PAYLOAD        pf);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Free request index.
pf	Pointer to the callback.

Return value

= IP_COAP_RETURN_OK Success.
≠ IP_COAP_RETURN_OK Error.

Refer to *IP_COAP_CLIENT_SetCommand* on page 1065 for usage example.

34.7.2.10 IP_COAP_CLIENT_SetReplyWaitTime()

Description

Configures the time to wait for the reply after sending the request. This is intended when using multicast/broadcast request.

Prototype

```
int IP_COAP_CLIENT_SetReplyWaitTime(IP_COAP_CLIENT_CONTEXT * pContext,
                                    unsigned                 Index,
                                    U32                      Seconds);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Free request index.
Seconds	Number of seconds to wait for the reply.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

When sending a request to a multicast/broadcast address, the type should be set to NON. The server should wait a random time before sending the reply (up to DEFAULT_LEISURE). The random time could be bigger than the normal wait time. This function is there to alter the wait time in this specific case.

34.7.2.11 IP_COAP_CLIENT_BuildAndSend()

Description

Builds the CoAP message and sends it based on a configured request.

Prototype

```
int IP_COAP_CLIENT_BuildAndSend(IP_COAP_CLIENT_CONTEXT * pContext,
                                unsigned                 Index);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Configured request index to build and send.

Return value

≥ 0 Success.
< 0 Error.

Refer to *IP_COAP_CLIENT_SetCommand* on page 1065 for usage example.

34.7.2.12 IP_COAP_CLIENT_GetLastResult()

Description

Returns the status and information when a request is completed.

Prototype

```
int IP_COAP_CLIENT_GetLastResult(IP_COAP_CLIENT_CONTEXT  * pContext,
                                 U8                      * pCode,
                                 U8                     ** ppError,
                                 U16                     * pLength);
```

Parameters

Parameter	Description
pContext	Client context.
pCode	To be filled with the CoAP code of the result.
ppError	In case of error sent by the server (class 2 or 4), it may contain a string with a comment on the error. To be ignored in case of success or other error classes.
pLength	Length of the ppError.

Return value

≥ 0 Index of the completed request.
< 0 No request just finished.

Additional information

When a request is sent, it is needed to call periodically IP_COAP_CLIENT_process() and IP_COAP_CLIENT_GetLastResult() to process the request and check when it is completed.

The data pointed by ppError may not be valid anymore after another received or sent message (case of IP_COAP_NSTART > 1).

Refer to *Sample CoAP client application* on page 1035 for usage example.

Warning

Observer result (apart from the abort) are transparent to this function. This means the return value will remain to -1 (nothing on-going) even when the observer request is completed, but the function needs to be called anyway in this case.

34.7.2.13 IP_COAP_CLIENT_GetMsgBuffer()

Description

Returns the buffer used to send messages.

Prototype

```
U8 *IP_COAP_CLIENT_GetMsgBuffer(IP_COAP_CLIENT_CONTEXT * pContext,
                                U16                    * pMsgLength);
```

Parameters

Parameter	Description
pContext	Client context.
pMsgLength	Value to fill with the buffer length.

Return value

≠ NULL Message buffer.
= NULL Error.

34.7.2.14 IP_COAP_CLIENT_GetLocationPath()

Description

Returns the Location-Path received. This should be call when processing reply of a POST for example.

Prototype

```
int IP_COAP_CLIENT_GetLocationPath(IP_COAP_CLIENT_CONTEXT  * pContext,
                                   unsigned                  Index,
                                   U8                     ** ppLoc,
                                   U8                      * pLocLength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
ppLoc	Pointer to place at the path start.
pLocLength	Value to fill with the path length.

Return value

= IP_COAP_RETURN_OK	Location-Path is present.
≠ IP_COAP_RETURN_OK	Error.

Additional information

The data pointed by the location pointer will not be valid after another received or sent message.

34.7.2.15 IP_COAP_CLIENT_GetLocationQuery()

Description

Returns the Location-Query received. This should be call when processing reply of a POST for example.

Prototype

```
int IP_COAP_CLIENT_GetLocationQuery(IP_COAP_CLIENT_CONTEXT  * pContext,
                                    unsigned                  Index,
                                    U8                     ** ppQuery,
                                    U8                      * pQueryLength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
ppQuery	Pointer to place at the query start.
pQueryLength	Value to fill with the Query length.

Return value

= IP_COAP_RETURN_OK	Location-Query is present.
≠ IP_COAP_RETURN_OK	Error.

Additional information

The data pointed by the query will not be valid after another message is received or sent.

34.7.2.16 IP_COAP_CLIENT_SetOptionURIPath()

Description

Adds the URI-Path option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionURIPath(IP_COAP_CLIENT_CONTEXT * pContext,
                                    unsigned                 Index,
                                    U8                     * pURI,
                                    U8                       URILength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
pURI	Start of the URI-Path (without the first '/').
URILength	Length of the URI-Path.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Only the pointer is kept, thus the passed data should be valid at least during the life of the request.

The given string for the URI-Path contains the complete URI-Path with '/' except for the first one. The break-down in many URI-Path options is done internally.

34.7.2.17 IP_COAP_CLIENT_SetOptionURIHost()

Description

Adds the URI-Host option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionURIHost(IP_COAP_CLIENT_CONTEXT * pContext,
                                    unsigned                 Index,
                                    U8                     * pHost,
                                    U8                       HostLength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
pHost	Start of the host name.
HostLength	Length of the host name.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Only the pointer is kept, thus the passed data should be valid at least during the life of the request.

34.7.2.18 IP_COAP_CLIENT_SetOptionURIPort()

Description

Adds the URI-Port option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionURIPort(IP_COAP_CLIENT_CONTEXT * pContext,
                                    unsigned                 Index,
                                    U16                      Port);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
Port	URI-Port value.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

34.7.2.19 IP_COAP_CLIENT_SetOptionURIQuery()

Description

Adds the URI-Query option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionURIQuery(IP_COAP_CLIENT_CONTEXT * pContext,
                                     unsigned                 Index,
                                     U8                     * pQuery,
                                     U8                       QueryLength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
pQuery	Start of the query.
QueryLength	Length of the query.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Only the pointer is kept, thus the passed data should be valid at least during the life of the request.

The given string for the URI-Query contains the complete URI-Query. The break-down in many URI-Query options is done internally using the character '&' as separators.

34.7.2.20 IP_COAP_CLIENT_SetOptionETag()

Description

Adds the ETag option to a request. It’S possible to add only one ETag per request.

Prototype

```
int IP_COAP_CLIENT_SetOptionETag(IP_COAP_CLIENT_CONTEXT * pContext,
                                 unsigned                 Index,
                                 U8                     * pETag,
                                 U8                       ETagLength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
pETag	Start of the ETag.
ETagLength	Length of the ETag.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Only the pointer is kept, thus the passed data should be valid at least during the life of the request.

34.7.2.21 IP_COAP_CLIENT_SetOptionBlock()

Description

Adds the block option to a request. Block1 of a request is a PUT or a POST, Block2 for a GET.

Prototype

```
int IP_COAP_CLIENT_SetOptionBlock(IP_COAP_CLIENT_CONTEXT * pContext,
                                  unsigned                 Index,
                                  U16                      Size);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
Size	Size of the block in bytes. 0 for default size.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

It is possible to give 0 as Size. The default block size will then be used.

Per default, GET requests don't have the Block2 option configured. If a reply is received with blocks, then the default block size is used. This function could be used to send the request with the Block2 option already set. In case of PUT/POST, Block1 option is always added.

34.7.2.22 IP_COAP_CLIENT_SetOptionAccept()

Description

Adds the Accept option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionAccept(IP_COAP_CLIENT_CONTEXT * pContext,
                                   unsigned                 Index,
                                   U16                      Value);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
Value	Accept format to be set in the request.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Some format are already defined in IP_COAP.h but any numerical value defined in RFCs could be used.

34.7.2.23 IP_COAP_CLIENT_SetOptionContentFormat()

Description

Adds the Content-Format option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionContentFormat(IP_COAP_CLIENT_CONTEXT * pContext,
                                          unsigned                 Index,
                                          U16                      Value);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
Value	Content format to be set in the request.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Some format are already defined in IP_COAP.h but any numerical value defined in RFCs could be used.

34.7.2.24 IP_COAP_CLIENT_SetOptionIfNoneMatch()

Description

Adds the If-None-Match option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionIfNoneMatch(IP_COAP_CLIENT_CONTEXT * pContext,
                                        unsigned                 Index);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

34.7.2.25 IP_COAP_CLIENT_SetOptionLocationPath()

Description

Adds the location path option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionLocationPath(IP_COAP_CLIENT_CONTEXT * pContext,
                                         unsigned                 Index,
                                         U8                     * pLocation,
                                         U8                       LocationLength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
pLocation	Pointer to the path.
LocationLength	Length of the path.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Only the pointer is kept, thus the passed data should be valid at least during the life of the request.

This function is provided even though the location path option appears normally only in replies (to POST requests).

The given string for the Location-Path contains the complete Location-Path with '/' except for the first one. The break-down in many Location-Path options is done internally.

34.7.2.26 IP_COAP_CLIENT_SetOptionLocationQuery()

Description

Adds the location query option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionLocationQuery(IP_COAP_CLIENT_CONTEXT * pContext,
                                          unsigned                 Index,
                                          U8                     * pQuery,
                                          U8                       QueryLength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
pQuery	Pointer to the query.
QueryLength	Length of the query.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Only the pointer is kept, thus the passed data should be valid at least during the life of the request.

This function is provided even though the location query option appears normally only in replies (to POST requests).

The given string for the Location-Query contains the complete Location-Query. The break-down in many Location-Query options is done internally using the character '&' as separators.

34.7.2.27 IP_COAP_CLIENT_SetOptionProxyURI()

Description

Adds the proxy URI option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionProxyURI(IP_COAP_CLIENT_CONTEXT * pContext,
                                     unsigned                 Index,
                                     U8                     * pURI,
                                     U16                      URILength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
pURI	Pointer to the URI.
URILength	Length of the URI.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Only the pointer is kept, thus the passed data should be valid at least during the life of the request.

34.7.2.28 IP_COAP_CLIENT_SetOptionProxyScheme()

Description

Adds the proxy scheme option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionProxyScheme(IP_COAP_CLIENT_CONTEXT * pContext,
                                        unsigned                 Index,
                                        U8                     * pScheme,
                                        U8                       SchemeLength);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
pScheme	Pointer to the scheme.
SchemeLength	Length of the scheme.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

Only the pointer is kept, thus the passed data should be valid at least during the life of the request.

34.7.2.29 IP_COAP_CLIENT_SetOptionSize1()

Description

Adds the Size1 option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionSize1(IP_COAP_CLIENT_CONTEXT * pContext,
                                  unsigned                 Index,
                                  U32                      Value);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
Value	Value of the Size1 option.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

34.7.2.30 IP_COAP_CLIENT_SetOptionAddIFMatch()

Description

Adds the If-Match option to a request.

Prototype

```
int IP_COAP_CLIENT_SetOptionAddIFMatch(IP_COAP_CLIENT_CONTEXT * pContext,
                                       unsigned                 Index,
                                       IP_COAP_IF_MATCH_INFO  * pIFMatch);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to configure.
pIFMatch	Parameter of the If-Match option.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

In a CoAP message, there could me many If-Match option. This option could indicate an ETag to match. It's also possible to have a "match all" with an empty ETag. To add a "match all", either provide a pIFMatch structure with the ETagLength set to 0, or give a NULL parameter as pIFMatch.

Only the pointer is kept, thus the passed data should be valid at least during the life of the request.

Example

```
static IP_COAP_IF_MATCH_INFO   IfMatch0;
static IP_COAP_IF_MATCH_INFO   IfMatch1;
static IP_COAP_IF_MATCH_INFO   IfMatch2;

//
// Send a PUT if the current ETag of "test" matches
// one of the three given ETag.
//
IP_COAP_CLIENT_SetCommand(&_COAPClient, Index, IP_COAP_TYPE_CON, IP_COAP_CODE_REQ_PUT);
IP_COAP_CLIENT_SetOptionURIPath(&_COAPClient, Index, "test", 4);
IfMatch0.ETagLength = 2;
IfMatch0.pETag      = "\x02\x03";
IP_COAP_CLIENT_SetOptionAddIFMatch(&_COAPClient, Index, &IfMatch0);
IfMatch1.ETagLength = 2;
IfMatch1.pETag      = "\x01\x02";
IP_COAP_CLIENT_SetOptionAddIFMatch(&_COAPClient, Index, &IfMatch1);
IfMatch2.ETagLength = 2;
IfMatch2.pETag      = "\x03\x03";
IP_COAP_CLIENT_SetOptionAddIFMatch(&_COAPClient, Index, &IfMatch2);
IP_COAP_CLIENT_SetOptionBlock(&_COAPClient, Index, 128);
IP_COAP_CLIENT_SetPayloadHandler(&_COAPClient, Index, _PUT_Handler);
//
IP_COAP_CLIENT_BuildAndSend(&_COAPClient, Index);
```


34.7.2.31 IP_COAP_CLIENT_OBS_Init()

Description

Initializes an observe request.

Prototype

```
int IP_COAP_CLIENT_OBS_Init(IP_COAP_CLIENT_CONTEXT * pContext,
                            unsigned                 Index,
                            IP_COAP_CLIENT_OBS     * pObs,
                            unsigned                 AutoToken);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Index of the free request to use to send the registration.
pObs	Observer to initializes.
AutoToken	Generates a token for the observe request.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

Additional information

An observe request must have a token. Either use this simple AutoToken mechanism or provide a token using IP_COAP_CLIENT_SetToken(). Note that the AutoToken is based on the current time. Thus creating two observe request at the same time will generate the same token for both requests and create issues.

Example

```
//
// Observe specific initialization.
//
IP_COAP_CLIENT_OBS_Init(&_COAPClient, Index, &_Observer, 0);
IP_COAP_CLIENT_OBS_SetEndCallback(&_Observer, _OBS_EndHandler, (void*)(U32)Index);
//
// Regular request initialization. Observe uses a GET request.
//
IP_COAP_CLIENT_SetCommand(&_COAPClient, Index, IP_COAP_TYPE_CON, IP_COAP_CODE_REQ_GET);
IP_COAP_CLIENT_SetOptionURIPath(&_COAPClient, Index, (U8*)"obs", 3);
IP_COAP_CLIENT_SetOptionBlock(&_COAPClient, Index, 16);
IP_COAP_CLIENT_SetPayloadHandler(&_COAPClient, Index, _GET_Handler);
//
// Token is mandatory for an observe, so either use the AutoToken from the Init or
// provide one.
//
IP_COAP_CLIENT_SetToken(&_COAPClient, Index, "\x01\x02\x03\x04", 4);
//
r = IP_COAP_CLIENT_BuildAndSend(&_COAPClient, Index);
```


34.7.2.32 IP_COAP_CLIENT_OBS_Abort()

Description

Stops an active observer.

Prototype

```
int IP_COAP_CLIENT_OBS_Abort(IP_COAP_CLIENT_CONTEXT * pContext,
                             unsigned                 Index,
                             IP_COAP_CLIENT_OBS     * pObs,
                             unsigned                 TryActiveAbort);
```

Parameters

Parameter	Description
pContext	Client context.
Index	Available request index to send the abort. Unused if TryActiveAbort is set to 0.
pObs	Observer to abort.
TryActiveAbort	Flag (0 or 1) used to prepare pObs to send a request to abort the registration.

Return value

IP_COAP_RETURN_OK	Success.
IP_COAP_RETURN_NOT_ALLOWED	Abort done, no request available for active abort.
Other	Error.

Example

Either perform a "lazy" abort. The clients will only remove the observer from its context. When the server will send a notification, the client will automatically reply with an error.

```
//
// Abort the observe (no abort sent to the server).
//
IP_COAP_CLIENT_OBS_Abort(&_COAPClient, 0 /* unused */, &_Observer, 0);
```

Or perform an active abort by sending a GET with the Observe option set to 1. The function is already setting the Observe value. Note that this abort request will be returned by IP_COAP_CLIENT_GetLastResult() contrarily to other observer related messages.

```
//
// Abort the observe indicating an abort will be sent.
//
IP_COAP_CLIENT_OBS_Abort(&_COAPClient, Index, &_Observer, 1);
//
// And actually send the abort reusing the on-going request.
//
IP_COAP_CLIENT_SetCommand(&_COAPClient, Index, IP_COAP_TYPE_CON, IP_COAP_CODE_REQ_GET);
IP_COAP_CLIENT_SetOptionURIPath(&_COAPClient, Index, (U8*)"obs", 3);
IP_COAP_CLIENT_SetPayloadHandler(&_COAPClient, Index, _GET_Handler);
//
r = IP_COAP_CLIENT_BuildAndSend(&_COAPClient, Index);
```


34.7.2.33 IP_COAP_CLIENT_OBS_SetEndCallback()

Description

Configures the end callback for an observe request. This callback is called after every transfer.

Prototype

```
int IP_COAP_CLIENT_OBS_SetEndCallback(IP_COAP_CLIENT_OBS  * pObs,
                                      PF_OBS_END_TRANSFER   pfObsEndTransfer,
                                      void                * pParam);
```

Parameters

Parameter	Description
pObs	Observer to configure.
pfObsEndTransfer	Callback called at the end of every transfer.
pParam	Parameter given to the callback. No internal usage.

Return value

= IP_COAP_RETURN_OK	Success.
≠ IP_COAP_RETURN_OK	Error.

34.7.3 Utility

These utility functions are helper functions to check various CoAP options. They are designed to be called from the payload callbacks.

34.7.3.1 IP_COAP_CheckAcceptFormat()

Description

Verifies if the Accept-Format matches the requested one.

Prototype

```
int IP_COAP_CheckAcceptFormat(IP_COAP_CALLBACK_PARAM * pParam,
                              U16                      Format);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
Format	Requested format.

Return value

IP_COAP_RETURN_OK	Accept-Format option value is not present or the Accept-Format matches the requested one.
IP_COAP_RETURN_ERR	Accept-Format option doesn't match.

34.7.3.2 IP_COAP_GetAcceptFormat()

Description

Reads the Accept-Format option value.

Prototype

```
int IP_COAP_GetAcceptFormat(IP_COAP_CALLBACK_PARAM * pParam,
                            U16                    * pFormat);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
pFormat	Variable to store the Accept-Format value.

Return value

IP_COAP_RETURN_OK	Accept-Format option value is present and set in pFormat.
IP_COAP_RETURN_ERR	Accept-Format option not present.

34.7.3.3 IP_COAP_CheckContentFormat()

Description

Verifies if the Content-Format match the requested one.

Prototype

```
int IP_COAP_CheckContentFormat(IP_COAP_CALLBACK_PARAM * pParam,
                               U16                      Format,
                               unsigned                 OptionMandatory);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
Format	Requested Content-Format.
OptionMandatory	Flag to indicate if the presence of the Content-Format option is mandatory or not. • 1: Mandatory. • 0: Optional.

Return value

IP_COAP_RETURN_OK	The option format matches the requested one or the option is not present (and the flag Option-Mandatory is 0).
IP_COAP_RETURN_OPTION_ERROR	Option is not present (OptionMandatory is 1)
IP_COAP_RETURN_ERR	Content-Format doesn't match the requested one.

34.7.3.4 IP_COAP_GetContentFormat()

Description

Reads the Content-Format option value.

Prototype

```
int IP_COAP_GetContentFormat(IP_COAP_CALLBACK_PARAM * pParam,
                             U16                    * pFormat);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
pFormat	Variable to store the Content-Format value.

Return value

IP_COAP_RETURN_OK	Content-Format option value is present and set in pFormat.
IP_COAP_RETURN_ERR	Content-Format option not present.

34.7.3.5 IP_COAP_IsLastBlock()

Description

Checks if the current block is the last one of the transfer.

Prototype

```
int IP_COAP_IsLastBlock(IP_COAP_CALLBACK_PARAM * pParam,
                        U8                       Code);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
Code	Code of the procedure • IP_COAP_CODE_REQ_GET • IP_COAP_CODE_REQ_PUT / IP_COAP_CODE_REQ_POST

Return value

1	This is the last block (or no block transfer)
0	More blocks are expected.

Additional information

If the procedure is a GET, Block2 is checked. Otherwise Block1 is checked.

34.7.3.6 IP_COAP_GetURIHost()

Description

Sets a pointer on the start of the URI-HostName.

Prototype

```
int IP_COAP_GetURIHost(IP_COAP_CALLBACK_PARAM  * pParam,
                       U8                     ** ppHost,
                       U8                      * pHostLength);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
ppHost	Address of the pointer where the HostName starts.
pHostLength	To be filled with the actual Host length.

Return value

IP_COAP_RETURN_OK	URI-Host option is present and ppHost, pHostLength are updated.
IP_COAP_RETURN_ERR	URI-Host option not present.

Additional information

If you plan on using the string later, make a copy of it as the pointer won't be valid after the callbacks.

34.7.3.7 IP_COAP_GetURIPath()

Description

Sets a pointer on the start of the URI-Path.

Prototype

```
int IP_COAP_GetURIPath(IP_COAP_CALLBACK_PARAM  * pParam,
                       U8                     ** ppURI,
                       U8                      * pURILength);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
ppURI	Address of the pointer where the URI starts.
pURILength	To be filled with the actual URI length.

Return value

IP_COAP_RETURN_OK	URI-Path option is present and ppURI, pURILength are updated.
IP_COAP_RETURN_ERR	URI-Path option not present.

Additional information

It makes only sense to be used by the server. In this case, the URI is re-assembled with '/' (the first one is omitted). For example if two URI-Path options are present with "sensor" and "temp", the returned string will be "sensor/temp".

If you plan on using the string later, make a copy of it as the pointer won't be valid after the callbacks.

34.7.3.8 IP_COAP_GetURIPort()

Description

Reads the URI-Port option value.

Prototype

```
int IP_COAP_GetURIPort(IP_COAP_CALLBACK_PARAM * pParam,
                       U16                    * pURIPort);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
pURIPort	Variable to store the URI-Port value.

Return value

IP_COAP_RETURN_OK	URI-Port option value is present and set in pURIPort.
IP_COAP_RETURN_ERR	URI-Port option not present.

34.7.3.9 IP_COAP_GetQuery()

Description

Sets a pointer on the start of the Query field.

Prototype

```
int IP_COAP_GetQuery(IP_COAP_CALLBACK_PARAM  * pParam,
                     U8                     ** ppQuery,
                     U16                     * pQueryLength);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
ppQuery	Address of the pointer where the Query starts.
pQueryLength	To be filled with the actual Query length.

Return value

IP_COAP_RETURN_OK	Query option is present and ppQuery, pQueryLength are updated.
IP_COAP_RETURN_ERR	Query option not present.

Additional information

If you plan on using the string later, make a copy of it as the pointer won't be valid after the callbacks.

34.7.3.10 IP_COAP_GetETag()

Description

Sets a pointer on the ETag value.

Prototype

```
int IP_COAP_GetETag(IP_COAP_CALLBACK_PARAM  * pParam,
                    U8                     ** ppETag,
                    U8                      * pETagLength);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
ppETag	Address of the pointer where the ETag starts.
pETagLength	To be filled with the actual ETag length.

Return value

IP_COAP_RETURN_OK	ETag option is present and ppETag, pETagLength are updated.
IP_COAP_RETURN_ERR	ETag option not present.

Additional information

If you plan on using the string later, make a copy of it as the pointer won't be valid after the callbacks.

34.7.3.11 IP_COAP_GetMaxAge()

Description

Reads the MaxAge option value.

Prototype

```
int IP_COAP_GetMaxAge(IP_COAP_CALLBACK_PARAM * pParam,
                      U32                    * pMaxAge);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
pMaxAge	Variable to store the MaxAge value.

Return value

IP_COAP_RETURN_OK	MaxAge option value is present and set in pMaxAge.
IP_COAP_RETURN_ERR	MaxAge option not present.

34.7.3.12 IP_COAP_GetSize1()

Description

Reads the Size1 option value.

Prototype

```
int IP_COAP_GetSize1(IP_COAP_CALLBACK_PARAM * pParam,
                     U32                    * pSize1);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
pSize1	Variable to store the Size1 value.

Return value

IP_COAP_RETURN_OK	Size1 option value is present and set in pSize1.
IP_COAP_RETURN_ERR	Size1 option not present.

34.7.3.13 IP_COAP_GetSize2()

Description

Reads the Size2 option value.

Prototype

```
int IP_COAP_GetSize2(IP_COAP_CALLBACK_PARAM * pParam,
                     U32                    * pSize2);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
pSize2	Variable to store the Size2 value.

Return value

IP_COAP_RETURN_OK	Size2 option value is present and set in pSize2.
IP_COAP_RETURN_ERR	Size2 option not present.

34.7.3.14 IP_COAP_GetLocationPath()

Description

Sets a pointer on the start of the Location-Path.

Prototype

```
int IP_COAP_GetLocationPath(IP_COAP_CALLBACK_PARAM  * pParam,
                            U8                     ** ppLoc,
                            U8                      * pLocLength);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
ppLoc	Address of the pointer where the location starts.
pLocLength	To be filled with the actual URI length.

Return value

IP_COAP_RETURN_OK	Location option is present and ppLoc, pLocLength are updated.
IP_COAP_RETURN_ERR	Location option not present.

Additional information

If you plan on using the string later, make a copy of it as the pointer won't be valid after the callbacks.

34.7.3.15 IP_COAP_GetLocationQuery()

Description

Sets a pointer on the start of the Location-Query field.

Prototype

```
int IP_COAP_GetLocationQuery(IP_COAP_CALLBACK_PARAM  * pParam,
                             U8                     ** ppQuery,
                             U16                     * pQueryLength);
```

Parameters

Parameter	Description
pParam	Pointer to the callback's parameter structure.
ppQuery	Address of the pointer where the Query starts.
pQueryLength	To be filled with the actual Query length.

Return value

IP_COAP_RETURN_OK	Query option is present and ppQuery, pQueryLength are updated.
IP_COAP_RETURN_ERR	Query option not present.

Additional information

If you plan on using the string later, make a copy of it as the pointer won't be valid after the callbacks.

34.8 Data structures

Structure / Callback	Description
Server	
IP_COAP_SERVER_CONTEXT	Main context of the server.
IP_COAP_SERVER_DATA	Context to describe a resource.
PF_POST_HANDLER	Server callback to create a new resource when receiving a POST.
pfGETPayload	Server data callback to fill the payload of a GET request.
pfPUTPayload	Server data callback to store the payload of a PUT request.
pfDELHandler	Server data callback to check the authorization to perform a DELETE request.
Client	
IP_COAP_CLIENT_CONTEXT	Main context of the client.
PF_OBS_END_TRANSFER	Observer end-of-transfer indication.
PF_CLIENT_PAYLOAD	Callback to handle the payload of GET/PUT.
Common	
IP_COAP_API	Contains the functions to perform the UDP receive and send as well as a function to get the current time in ms.
IP_COAP_CALLBACK_PARAM	Argument of the payload callbacks describing connection parameters.
IP_COAP_OPTIONS_INFO	CoAP options description.
IP_COAP_IF_MATCH_INFO	Utility structure to parametrize the If-Match option.
IP_COAP_HEADER_INFO	Parsed CoAP header.
IP_COAP_BLOCK_INFO	Definition of the block: index and size.
IP_COAP_CONN_INFO	UDP connection definition.
pfReceive	Callback used to receive UDP packets.
pfSend	Callback used to send UDP packets.
pfGetTimeMs	Callback used to get the current time in ms.

34.8.1 Structure IP_COAP_SERVER_CONTEXT

Description

Main context of the server.

Prototype

```
struct {
  //
  // Buffer for the UDP packet handling (Rx/Tx).
  //
  U8*                           pMsgBuffer;
  U16                           MsgBufferSize;
  //
  // APIs for UDP transmission and time.
  //
  int                           (*pfReceive)  (U8*                 pBuffer,
                                               unsigned            BufferSize,
                                               IP_COAP_CONN_INFO* pInfo,
                                               unsigned*           pIsMulticast);
  int                           (*pfSend)     (U8*                 pBuffer,
                                               unsigned            BufferSize,
                                               IP_COAP_CONN_INFO* pInfo);
  U32                           (*pfGetTimeMs)(void);
  //
  // Server parameters.
  //
  const char*                   sHostName;
  const char*                   sErrorDesc;
  IP_COAP_SERVER_DATA*          pFirstData;
  IP_COAP_SERVER_DATA           DataWellKnownCore;
  //
  IP_COAP_SERVER_CLIENT_INFO*   pFirstClient;
  IP_COAP_SERVER_CLIENT_INFO*   pFreeCLient;
  //
  IP_COAP_OBSERVER*             pFirstObs;
  IP_COAP_OBSERVER*             pFreeObs;
  //
  PF_POST_HANDLER               pfPOSTCreateEntry;
  //
  // Internal data.
  //
  IP_COAP_SERVER_DATA*          pDataFirstFound;
  U8*                           pPayload;
  U32                           OptionMask;
  U16                           UDPPort;
  U16                           PayloadLength;
  U16                           StructureETag;
  U16                           URILengthReq;
  U16                           MsgLength;
  U16                           BadOption;
  U16                           MessageId;
  U8                            DefaultSzx;
  U8                            ConfigMask;
};
```

Member	Description
pMsgBuffer	Buffer used to receive and send CoAP messages in UDP. Initialized in IP_COAP_SERVER_Init()
MsgBufferSize	Size of the message buffer.
pfReceive	Callback to receive UDP packets (see *Callback pfReceive* on page 1135)

Member	Description
pfSend	Callback to send UDP packets (see *Callback pfSend* on page 1135)
pfGetTimeMs	Callback to get the current time in milliseconds (see *Callback pfGetTimeMs* on page 1135).
sHostName	If it is requested to send the Uri-Host option by a IP_COAP_SERVER_DATA , the hostname used is this one. (initialized to a null pointer). All resources associated to a server are using the same host name. This is a null terminated string.
sErrorDesc	Null terminated string sent in an error message.
pFirstData	Start of the linked list of server resources. Used by IP_COAP_SERVER_AddData() and IP_COAP_SERVER_RemoveData().
DataWellKnownCore	Resource used to reply to a discover (GET .well-known/core). This will automatically return the list of all resources of the server with the CoRE format.
pFirstClient	Start of the linked list of the currently connected clients. Internal usage only.
pFreeCLient	Linked list of the client structure pool. Initialized with IP_COAP_SERVER_AddClientBuffer().
pFirstObs	Start of the linked list of the currently connected observers. Internal usage only.
pFreeObs	Linked list of the observer structure pool. Initialized with IP_COAP_SERVER_AddObserverBuffer().
pfPOSTCreateEntry	Call back used to create a new entry when getting a POST command. Initialized with IP_COAP_SERVER_SetPOSTHandler(). If left unset (NULL), the server won't allow the creation of new resource entries and will automatically reply with a 4.05 Method Not Allowed.
UDPPort	If it is requested to send the Uri-Port option by a IP_COAP_SERVER_DATA , the port used is this one (initialized to the default port). All resources associated to a server are using the same port.
Other entries of this structure are for internal usage only and are not described in this document.	

Additional information

The fields of this structure should be set only via APIs or internal functions.

A client pool is given to the server with the function IP_COAP_SERVER_AddClientBuffer(). The size of this pool defines the number of clients connected to the server at the same time. The server could handle more clients as the connections are closed once data transfers are completed. Thus a buffer of at least one client is mandatory but even in this case, many more actual clients could be served especially if there are no long block transfers and CON is used to mitigate congestions.

An observer pool is given to the server with the function IP_COAP_SERVER_AddObserverBuffer(). The size of this pool defines the number of observers handled by the server at the same time. It could be null if no observable resources are configured.

34.8.2 Structure IP_COAP_SERVER_DATA

Description

Server context to describe a resource.

Prototype

```
struct IP_COAP_SERVER_DATA_STRUCT {
  IP_COAP_SERVER_DATA*    pNext;
  //
  const char*             sURI;
  const char*             sDescription;
  U8                      ETag[8];
  U32                     MaxAge;
  U32                     Size2;
  U32                     DefGetOptMask;
  U16                     ContentFormat;
  U8                      ETagLength;
  U8                      ObsConfig;
  //
  // Callbacks to handle the requests.
  //
  int                     (*pfGETPayload)(IP_COAP_SERVER_CONTEXT* pContext,
                                          U8**                    ppBuffer,
                                          U16*                    pLength,
                                          IP_COAP_CALLBACK_PARAM* pParam);
  int                     (*pfPUTPayload)(IP_COAP_SERVER_CONTEXT* pContext,
                                          U8*                     pPayload,
                                          U16                     Length,
                                          IP_COAP_CALLBACK_PARAM* pParam);
  int                     (*pfDELHandler)(IP_COAP_SERVER_CONTEXT* pContext,
                                          IP_COAP_CALLBACK_PARAM* pParam);
};
```

Member	Description
pNext	Anchor for the linked list. Internal use only.
sURI	Null terminated string for the resource Uri-Path. The first '/' is omitted.
sDescription	Null terminated string for the description of the resource in CoRE format. If the resource is an observable, "obs" is automatically added and should not be set here.
ETag	entity-tag (ETag) of the resource.
MaxAge	MaxAge (value validity) duration of a the resource after a GET.
Size2	Size of the resource sent in a GET reply if configured so.
DefGetOptMask	Default options characterizing the resource sent in a GET reply (see below).
ContentFormat	Content-Format of the resource. Some format are defined in the RFC and are in IP_COAP.h (IP_COAP_CT_xxx) but other numerical values could be used.
ETagLength	Length of ETag.
ObsConfig	Configuration of the observe parameters of the resource (see below). Set to 0 if the resource is not observable.
pfGETPayload	Callback used to fill the payload when replying to a GET (see pfGETPayload). If set to NULL, the server will reply with a 4.05 Not Allowed when receiving the request.

Member	Description
pfPUTPayload	Callback used to handle the payload when receiving a PUT or POST (see pfPUTPayload). If set to NULL, the server will reply with a 4.05 Not Allowed when receiving the request.
pfDELHandler	Callback used to check the authorization of a DELETE request (see pfDELHandler). If set to NULL, the server will reply with a 4.05 Not Allowed when receiving the request.

Additional information

This structure should be initialized with 0 (memset) and then configured before being added to the server with the function IP_COAP_SERVER_AddData().

The DefGetOptMask describes which options should be sent by default in a GET reply to inform the client. The mask format is defined by IP_COAP_OPTMASK_xxx defines. The possible options to set in this define are:

- IP_COAP_OPTMASK_ETAG:
  Send the entity-tag of the resource. ETag and ETagLength should be correctly set.
- IP_COAP_OPTMASK_URI_PORT:
  Send the UDP Port used to communicate with the resource. UDPPort of the server context IP_COAP_SERVER_CONTEXT should be correctly set.
- IP_COAP_OPTMASK_URI_HOST:
  Send the Uri-Host of the resource. sHostName of the server context IP_COAP_SERVER_CONTEXT should be correctly set.
- IP_COAP_OPTMASK_CONTENT_FORMAT:
  Send the content-format of the resource. ContentFormat should be correctly set.
- IP_COAP_OPTMASK_MAX_AGE:
  Send the max-age of the resource (validity time) in seconds. MaxAge should be correctly set.
- IP_COAP_OPTMASK_SIZE2:
  Send the data length of the resource (approximated value). This could be used by the client in case of block transfer. Size2 should be correctly set.

The observe configuration in ObsConfig is also a mask using the define IP_COAP_OBS_xxx. The bit IP_COAP_OBS_OBSERVABLE should always be set to indicate the resource is observable. It is possible to optionally configure either IP_COAP_OBS_AUTO_CON_ON_MAX_AGE or IP_COAP_OBS_AUTO_NON_ON_MAX_AGE. The server will then automatically send a notification (either CON or NON) every MaxAge seconds. For example, it is possible to send a NON notification at every MaxAge expiry and a CON notification when the value is actually changed (by calling IP_COAP_SERVER_UpdateData()).

Example

```
  pServerData                   = &_COAPServerData[0];
  IP_COAP_MEMSET(pServerData, 0, sizeof(IP_COAP_SERVER_DATA));
  //
  // URI-Path with the first '/' omitted.
  //
  pServerData->sURI             = "temperature/meas";
  //
  // Description: optional and could be set to NULL. Follows the CoRE format.
  // "obs" is added automatically is the resource is defined as observable.
  //
  pServerData->sDescription     = "ct=0;title=\"Temperature measurement done every 2
minutes\"";
  //
  // Callback to perform GET and PUT. DELETE is disabled as the callback
  // is not set.
  //
  pServerData->pfGETPayload     = _Temperature_Meas1_GetPayload;
  pServerData->pfPUTPayload     = _Temperature_Meas1_PutPayload;
  //
  // Default content format is plain/text.
  //
  pServerData->ContentFormat    = IP_COAP_CT_TXT;
```



```
  //
  // Configure an ETag.
  //
  IP_COAP_MEMCPY(&pServerData->ETag[0], "\x01\x02\x03\x04\x05\x06\x07\x08", 8);
  pServerData->ETagLength        = 8;
  //
  // Data are valid for 2 minutes.
  //
  pServerData->MaxAge            = 120;
  //
  // data size is approximately 8 bytes long.
  //
  pServerData->Size2             = 8;
  //
  // In a reply to a GET, send the options: ETag, Content-Format, Max-Age and Size2.
  //
  pServerData->DefGetOptMask     = IP_COAP_OPTMASK_ETAG | IP_COAP_OPTMASK_CONTENT_FORMAT |
                                   IP_COAP_OPTMASK_MAX_AGE | IP_COAP_OPTMASK_SIZE2;
  //
  // This resource is observable. Server will send a NON at Max-Age period.
  //
  pServerData->ObsConfig         = IP_COAP_OBS_OBSERVABLE | IP_COAP_OBS_AUTO_NON_ON_MAX_AGE;
```


34.8.3 Callback PF_POST_HANDLER

Description

Server callback to create a new resource when receiving a POST.

Prototype

```
typedef int (*PF_POST_HANDLER)(IP_COAP_SERVER_CONTEXT* pContext,
                               IP_COAP_CALLBACK_PARAM* pParam,
                               U32                     PayloadLength,
                               IP_COAP_SERVER_DATA**   ppServerData);
```

Member	Description
pContext	Pointer on the server context.
pParam	Parameter structure. See IP_COAP_CALLBACK_PARAM }.
PayloadLength	Indication on the payload length. It is either the Size1 parameter if given or the length of the 1st block.
ppServerData	Receives the created IP_COAP_SERVER_DATA . Don't call IP_COAP_SERVER_AddData() as it will be done by the server if the pointer is set.

Additional information

The return values are of type IP_COAP_RETURN. The possible values are:

- IP_COAP_RETURN_OK: Entry created.
- Any one of the defined error return values. If an error is returned, it could be usefull for the operator to add a short description (null-terminated string) that would be appended to the error message.

The actual payload will be handle by the PUT payload handler (pfPUTPayload) of the newly created resource. If an error happens the resource will be removed from the server context.

Example

```
static int _POSTCreateEntry(IP_COAP_SERVER_CONTEXT* pContext, IP_COAP_CALLBACK_PARAM* pParam,
                            U32 PayloadLength, IP_COAP_SERVER_DATA** ppServerData) {
  U8* pURI;
  U8  Length;

  //
  // Check the validity of the POST request.
  //
  if (POST_Is_Authorized) {
    IP_COAP_SERVER_SetErrorDescription(pContext, "Client not authorized");
    return IP_COAP_RETURN_NOT_ALLOWED;
  }
  //
  if (PayloadLength > POST_DATA_LENGTH) {
    IP_COAP_SERVER_SetErrorDescription(pContext, "Maximum size is 128 bytes");
    return IP_COAP_RETURN_BUFFER_TOO_SMALL;
  }
  //
  if (IP_COAP_GetURIPath(pParam, &pURI, &Length) != IP_COAP_RETURN_OK) {
    return IP_COAP_RETURN_ERR;
  }
  //
  if (Length >= 64) {
    IP_COAP_SERVER_SetErrorDescription(pContext, "Maximum URI path is 64 long");
    return IP_COAP_RETURN_BUFFER_TOO_SMALL;
  }
  //
  // Create the new data.
  // 1. Copy the Uri-Path in a static string.
  //
```



```
  IP_COAP_MEMCPY(&_POSTUriPath[0], pURI, Length);
  _POSTUriPath[Length]         = 0;  // Terminate the string with a '0'.
  //
  // 2. Initialize the new resource.
  //
  IP_COAP_MEMSET(&_ServerData, 0, sizeof(IP_COAP_SERVER_CLIENT_INFO));
  _POSTServerData.sURI         = &_POSTUriPath[0];
  _POSTServerData.sDescription = "title=\"User created entry\"";
  _POSTServerData.pfGETPayload = _POSTCreateEntry_GetPayload;
  _POSTServerData.pfPUTPayload = _POSTCreateEntry_PutPayload;
  _POSTServerData.pfDELHandler = _POSTCreateEntry_DelHandler;
  //
  // 3. Set the ServerData pointer. Don't call IP_COAP_SERVER_AddData(), it is done
  // by the server automatically.
  //
  *ppServerData = &_POSTServerData;

  return IP_COAP_RETURN_OK;
}
```


34.8.4 Callback pfGETPayload

Description

Server data callback to fill the payload of a GET request.

Prototype

```
int    (*pfGETPayload)(IP_COAP_SERVER_CONTEXT* pContext,
                       U8**                   ppBuffer,
                       U16*                   pLength,
                       IP_COAP_CALLBACK_PARAM* pParam);
```

Member	Description
pContext	Pointer on the server context.
ppBuffer	Start of the payload location. To be moved to the end of the copied data.
pLength	Length of the remaining bytes in the message/block. To be reduced from the number of added data.
pParam	Callback parameters. Refer to IP_COAP_CALLBACK_PARAM .

Additional information

The callback parameter contains information regarding the request, especially the block information.

The return values are of type IP_COAP_RETURN. The possible values are:

- IP_COAP_RETURN_SEND_END: The payload is added and this is the last block (or no block used).
- IP_COAP_RETURN_SEND_SEPARATE: The data are not ready, send an ACK. IP_COAP_SERVER_UpdateData() will be called later to reply to the client with a new request.
- IP_COAP_RETURN_SEND_BLOCK: The payload is added but there are more data to copy. The block should be fully filled.
- Any one of the defined error return values. If an error is returned, it could be useful for the operator to add a short description (null-terminated string) that would be appended to the error message.

Note: There are two other possible return values:

- IP_COAP_RETURN_IGNORE_BLOCK and IP_COAP_RETURN_IGNORE_END: These have the same behavior as IP_COAP_RETURN_SEND_BLOCK and IP_COAP_RETURN_SEND_END, but the server won't send the message currently built. It is up to the application to build and send the complete CoAP message on its own. It shouldn't be needed to use them.

Example

First example with a simple short data (no block). The data needs some time to be computed and thus sends a separate reply (first send a simple ACK and when the data is ready the application have to call IP_COAP_SERVER_UpdateData() which will call the callback again to send the reply).

```
static int _GetPayload(IP_COAP_SERVER_CONTEXT* pContext, U8** ppBuffer,
                       U16* pLength, IP_COAP_CALLBACK_PARAM* pParam) {
  U8* pBuffer;
  U16 Length;
  int r;

  (void)pContext;

  if (IP_COAP_CheckAcceptFormat(pParam, IP_COAP_CT_TXT) != IP_COAP_RETURN_OK) {
    IP_COAP_SERVER_SetErrorDescription(pContext, "plain/text only");
    return IP_COAP_RETURN_CT_FORMAT_ERROR;
```



```
  }
  //
  pBuffer = *ppBuffer;
  Length  = *pLength;
  //
  if (_IsDataReady() == 0) {
    //
    // Trigger the processing to get the data.
    // Ask to send an ACK in the meantime.
    //
    return IP_COAP_RETURN_SEND_SEPARATE;
  }
  //
  // Result is ready.
  //
  r = SEGGER_snprintf((char*)pBuffer, Length, "Data: %d", _Data);
  pBuffer += r;
  Length  -= r;
  //
  *ppBuffer = pBuffer;
  *pLength  = Length;

  return IP_COAP_RETURN_SEND_END;
}
```

Second example with block support. The resource data is in `_Data` and its length is in `_DataLength`. Reply is piggybacked with the ACK.

```
//
// The data stored in the variable _Data have a length of _DataLength.
//
static int _GetPayload(IP_COAP_SERVER_CONTEXT* pContext, U8** ppBuffer,
                       U16* pLength, IP_COAP_CALLBACK_PARAM* pParam) {
  U8* pBuffer;
  U16 Length;
  int Offset;
  int Len;
  int r;

  (void)pContext;

  //
  // If the content format option is present, check this is a plain/text.
  //
  if (IP_COAP_CheckAcceptFormat(pParam, IP_COAP_CT_TXT) != IP_COAP_RETURN_OK) {
    IP_COAP_SERVER_SetErrorDescription(pContext, "plain/text only");
    return IP_COAP_RETURN_CT_FORMAT_ERROR;
  }
  //
  pBuffer = *ppBuffer;
  Length  = *pLength;
  r       = IP_COAP_RETURN_SEND_BLOCK;  // By default indicates there are more data.
  Len     = Length;
  //
  if (pParam->pBlock != NULL) {
    Offset = pParam->pBlock->Index * pParam->pBlock->Size;
  } else {
    Offset = 0;
  }
  //
  if (Offset >= _DataLength) {
    return IP_COAP_RETURN_NO_PAYLOAD;
  }
  if ((Offset + Len) >= _DataLength) {
    //
    // Last block of the transfer.
    //
    Len = _DataLength - Offset;
    r   = IP_COAP_RETURN_SEND_END;
  }
  IP_COAP_MEMCPY(pBuffer, _Data + Offset, Len);
  pBuffer += Len;
  Length  -= Len;
  //
  *ppBuffer = pBuffer;
```



```
  *pLength  = Length;

  return r;
}
```


34.8.5 Callback pfPUTPayload

Description

Server data callback to store the payload of a PUT request.

Prototype

```
int    (*pfPUTPayload)(IP_COAP_SERVER_CONTEXT* pContext,
                       U8*                    pPayload,
                       U16                    Length,
                       IP_COAP_CALLBACK_PARAM* pParam);
```

Member	Description
pContext	Pointer on the server context.
pPayload	Start of the payload memory.
Length	Length of the payload.
pParam	Callback parameters. Refer to IP_COAP_CALLBACK_PARAM .

Additional information

The callback parameter contains information regarding the request, especially the block information.

The return values are of type IP_COAP_RETURN. The possible values are:

- IP_COAP_RETURN_OK: The resource is updated with the received data block.
- Any one of the defined error return values. If an error is returned, it could be useful for the operator to add a short description (null-terminated string) that would be appended to the error message. For example IP_COAP_RETURN_NO_PAYLOAD will trigger an error “4.08 Request Entity Incomplete”.

Example

The first example is a short data (no block support) accepting either plain/text or application/octet-stream.

```
static int _PutPayload(IP_COAP_SERVER_CONTEXT* pContext, U8* pPayload,
                       U16 Length, IP_COAP_CALLBACK_PARAM* pParam) {
  int r;
  U16 Format;

  if (Length == 0) {
    return IP_COAP_RETURN_ERR;
  }
  if (pParam->pBlock != NULL) {
    if (pParam->pBlock->Index > 0) {
      IP_COAP_SERVER_SetErrorDescription(pContext, "Block transfer not supported");
      return IP_COAP_RETURN_ERR;
    }
  }
  //
  // Check if there is a content format specified.
  //
  if (IP_COAP_GetContentFormat(pParam, &Format) != IP_COAP_RETURN_OK) {
    //
    // Assume plain/text by default if Content-Format option is not set.
    //
    Format = IP_COAP_CT_TXT;
  }
  //
  if (Format == IP_COAP_CT_TXT) {
    //
    // Format plain/text.
    //
    r = _ParseString(pPayload, Length);
```



```
    if (r != 0) {
      IP_COAP_SERVER_SetErrorDescription(pContext, "Invalid string format");
      return IP_COAP_RETURN_ERR;
    }
  } else if (Format == IP_COAP_CT_OCTET_STREAM) {
    //
    // Format binary.
    //
    r = _ParseBinary(pPayload, Length);
    if (r != 0) {
      IP_COAP_SERVER_SetErrorDescription(pContext, "Invalid binary format");
      return IP_COAP_RETURN_ERR;
    }
  } else {
    IP_COAP_SERVER_SetErrorDescription(pContext, "Accept either plain/text or octet-
stream");
    return IP_COAP_RETURN_CT_FORMAT_ERROR;
  }

  return IP_COAP_RETURN_OK;
}
```

Other example with block support.

```
static int _PutPayload(IP_COAP_SERVER_CONTEXT* pContext, U8* pPayload,
                       U16 Length, IP_COAP_CALLBACK_PARAM* pParam) {
  int Offset;

  //
  // Request that content format is present and of type plain/text.
  //
  if (IP_COAP_CheckContentFormat(pParam, IP_COAP_CT_TXT, 1) != IP_COAP_RETURN_OK) {
    IP_COAP_SERVER_SetErrorDescription(pContext, "Content Format plain/text requested");
    return IP_COAP_RETURN_CT_FORMAT_ERROR;
  }
  //
  if (pParam->pBlock != NULL) {
    Offset = pParam->pBlock->Index * pParam->pBlock->Size;
  } else {
    Offset = 0;
  }
  //
  if (Offset >= 128) {
    return IP_COAP_RETURN_BUFFER_TOO_SMALL;
  }
  if ((Offset + Length) > 128) {
    return IP_COAP_RETURN_BUFFER_TOO_SMALL;
  }
  IP_COAP_MEMCPY(_Data + Offset, pPayload, Length);
  //
  // Check if this is the end of the transfer.
  //
  if (IP_COAP_IsLastBlock(pParam, IP_COAP_CODE_REQ_PUT) != 0) {
    _FinishDataUpdate();
  }

  return IP_COAP_RETURN_OK;
}
```


34.8.6 Callback pfDELHandler

Description

Server data callback to check the authorization to perform a DELETE request.

Prototype

```
int    (*pfDELHandler)(IP_COAP_SERVER_CONTEXT* pContext,
                       IP_COAP_CALLBACK_PARAM* pParam);
```

Member	Description
pContext	Pointer to the server context.
pParam	Callback parameters. Refer to IP_COAP_CALLBACK_PARAM .

Additional information

The callback parameter contains information regarding the request, especially the client UDP connection info.

The return values are of type IP_COAP_RETURN. The possible values are:

- IP_COAP_RETURN_OK: The resource is deleted. Do not call IP_COAP_SERVER_RemoveData(). It will be done by the server.
- IP_COAP_RETURN_OK_NO_DELETE: The server will reply with a 2.02 Deleted but will not actually remove the data (no call to IP_COAP_SERVER_RemoveData()).
- Any one of the defined error return values. If an error is returned, it could be useful for the operator to add a short description (null-terminated string) that would be appended to the error message.

Example

```
static int _DelHandler(IP_COAP_SERVER_CONTEXT* pContext, IP_COAP_CALLBACK_PARAM* pParam) {

  (void)pContext;

  if (_IsDeleteAuthorized(pParam->pConnInfo) == 0) {
    return IP_COAP_RETURN_NOT_ALLOWED;
  }
  //
  // Data will be deleted, free some memory.
  //
  _FreeData();

  return IP_COAP_RETURN_OK;
}
```


34.8.7 Structure IP_COAP_CLIENT_CONTEXT

Description

Main context of the client. A client is dedicated to a server. If it is needed to have activities with two servers during the same period of time, then two clients are needed.

Prototype

```
struct {
  //
  // Buffer for the UDP packet handling (Rx/Tx).
  //
  U8*                          pMsgBuffer;
  U16                          MsgBufferSize;
  //
  // UDP transmission function.
  //
  int                          (*pfReceive)  (U8*                pBuffer,
                                              unsigned           BufferSize,
                                              IP_COAP_CONN_INFO* pInfo,
                                              unsigned*          pIsMulticast);
  int                          (*pfSend)     (U8*                pBuffer,
                                              unsigned           BufferSize,
                                              IP_COAP_CONN_INFO* pInfo);
  U32                          (*pfGetTimeMs)(void);
  //
  // Client parameters.
  //
  IP_COAP_CLIENT_REQUEST       aRequest[IP_COAP_NSTART];
  IP_COAP_CONN_INFO            ConnInfo;
  IP_COAP_CLIENT_OBS*          pFirstObs;
  //
  // Internal data.
  //
  U8*                          pPayload;
  U16                          PayloadLength;
  U16                          MessageId;
  U8                           DefaultSzx;
};
```

Member	Description
pMsgBuffer	Buffer used to receive and send CoAP messages in UDP. Initialized in IP_COAP_CLIENT_Init()
MsgBufferSize	Size of the message buffer.
pfReceive	Callback to receive UDP packets (see *Callback pfReceive* on page 1135)
pfSend	Callback to send UDP packets (see *Callback pfSend* on page 1135)
pfGetTimeMs	Callback to get the current time in milliseconds (see *Callback pfGetTimeMs* on page 1135).
aRequest	Structure used to describe a request to the server.
ConnInfo	UDP connection information of the server. Set by calling IP_COAP_CLIENT_SetServerAddress().
pFirstObs	Start of the linked list of the active observers. Modified through IP_COAP_CLIENT_OBS_Init() and IP_COAP_CLIENT_OBS_Abort().
Other entries of this structure are for internal usage only and are not described in this document.	

Additional information

This structure shouldn’t be directly modified. Only APIs should be used.

34.8.8 Callback PF_OBS_END_TRANSFER

Description

Observer end-of-transfer indication. This function is called by the client process whenever a notification from an observed data is totally received.

Prototype

```
typedef void (*PF_OBS_END_TRANSFER)(U8 Code, int IsFinal, void* pParam);
```

Member	Description
Code	CoAP code of the received notification.
IsFinal	Indication that the server canceled the observe, either because of some errors or because the server didn’t reply with the observe option.
pParam	Parameter given to the function IP_COAP_CLIENT_OBS_SetEndCallback(). It is unused by the the client process (application usage only).

Additional information

The server may “lose” the observer information without informing the client (due to loss of packets for example). This is the role of the application to refresh the observe registration if no notification is received for some time.

34.8.9 Callback PF_CLIENT_PAYLOAD

Description

Client callback to handle the payload of GET/PUT.

Prototype

```
typedef int    (*PF_CLIENT_PAYLOAD)(IP_COAP_CLIENT_CONTEXT* pContext,
                                    U8**                    ppPayload,
                                    U16*                    pLength,
                                    IP_COAP_CALLBACK_PARAM* pParam);
```

Member	Description
pContext	Pointer on the server context.
ppBuffer	Start of the payload location. To be moved to the end of the copied data.
pLength	Length of the remaining bytes in the message/block. To be reduced from the number of added data.
pParam	Callback parameters. Refer to IP_COAP_CALLBACK_PARAM .

Additional information

The return values are of type IP_COAP_RETURN. The possible values are:

- IP_COAP_RETURN_SEND_END: The payload is added to PUT or POST request and this is the last block (or no block used).
- IP_COAP_RETURN_SEND_BLOCK: The payload is added to PUT or POST request and there are more data to copy. The block should be fully filled.
- IP_COAP_RETURN_NO_PAYLOAD: For some reason, there are no payload to copy.
- IP_COAP_RETURN_OK: The payload from a GET reply is handled.
- Any one of the defined error return values. If an error is returned, it could be useful for the operator to add a short description (null-terminated string) that would be appended to the error message.

Examples

This first example is a callback for a GET request. The callback will be called when the response with the content is received. Separate responses are transparent for the client.

```
static int _GET_Handler(IP_COAP_CLIENT_CONTEXT* pContext, U8** ppPayload,
                        U16* pLength, IP_COAP_CALLBACK_PARAM* pParam) {
  int Offset;

  (void)pContext;
  //
  // Check the content-format if present.
  //
  if (IP_COAP_CheckContentFormat(pParam, IP_COAP_CT_TXT, 0) !=  IP_COAP_RETURN_OK) {
    return IP_COAP_RETURN_CT_FORMAT_ERROR;
  }
  //
  // Some init on the first block.
  //
  if (pParam->pBlock->Index == 0) {
    _InitData();
  }
  //
  // Copy the data block.
  //
  Offset = pParam->pBlock->Index * pParam->pBlock->Size;
  IP_COAP_MEMCPY(_Data + Offset, *ppPayload, *pLength);
  //
  // Perform some processing on the last block.
  //
```



```
  if (IP_COAP_IsLastBlock(pParam, IP_COAP_CODE_REQ_GET) != 0) {
    _HandleGetEnd();
  }

  return IP_COAP_RETURN_OK;
}
```

The second example is a callback for a PUT/POST request.

```
static int _PUT_Handler(IP_COAP_CLIENT_CONTEXT* pContext, U8** ppPayload,
                        U16* pLength, IP_COAP_CALLBACK_PARAM* pParam) {
  int Offset;
  U8* pBuf;
  U16 Length;
  U16 l;
  int r;

  (void)pContext;

  pBuf          = *ppPayload;
  Length        = *pLength;
  r             = IP_COAP_RETURN_SEND_END;          // By default, this is the last block.
  //
  Offset = pParam->pBlock->Index * pParam->pBlock->Size;
  if (Offset >= _PayloadLength) {
    _ErrorDetected = 1;
    return IP_COAP_RETURN_NO_PAYLOAD;
  }
  //
  // Compute the remaining length of the payload not sent yet.
  //
  l = _PayloadLength - Offset;
  //
  // If the remaining length of the payload is bigger than this
  // block size, reduce it to the block size.
  //
  if (l > Length) {
    r = IP_COAP_RETURN_SEND_BLOCK;  // More data to send in a next block.
    l = Length;
  }
  IP_COAP_MEMCPY(pBuf, _Payload + Offset, l);
  pBuf   += l;
  Length -= l;
  //
  *ppPayload = pBuf;
  *pLength   = Length;

  return r;
}
```


34.8.10 Structure IP_COAP_API

Description

Contains the functions to perform the UDP receive and send as well as a function to get the current time in ms.

Prototype

```
typedef struct {
  int   (*pfReceive)  (U8*                pBuffer,
                       unsigned           BufferSize,
                       IP_COAP_CONN_INFO* pInfo,
                       unsigned*          pIsMulticast);
  int   (*pfSend)     (U8*                pBuffer,
                       unsigned           BufferSize,
                       IP_COAP_CONN_INFO* pInfo);
  U32   (*pfGetTimeMs)(void);
} IP_COAP_API;
```

Member	Description
pfReceive	Callback to receive UDP packets (see *Callback pfReceive* on page 1135)
pfSend	Callback to send UDP packets (see *Callback pfSend* on page 1135)
pfGetTimeMs	Callback to get the current time in milliseconds (see *Callback pfGetTimeMs* on page 1135).

34.8.11 Structure IP_COAP_CALLBACK_PARAM

Description

Argument of the payload callbacks describing connection parameters.

Prototype

```
typedef struct {
  IP_COAP_BLOCK_INFO*    pBlock;
  IP_COAP_CONN_INFO*     pConnInfo;
  IP_COAP_HEADER_INFO*         pHeader;
  IP_COAP_OPTIONS_INFO*  pOptDesc;
} IP_COAP_CALLBACK_PARAM;
```

Member	Description
pBlock	Block information. See IP_COAP_BLOCK_INFO .
pConnInfo	UDP connection information. See IP_COAP_CONN_INFO .
pHeader	Parsed CoAP message header. See IP_COAP_HEADER_INFO .
pOptDesc	CoAP message option description. See IP_COAP_OPTIONS_INFO .

Warning

Always check that the pointer is valid (not NULL) before accessing one of the different fields as some might not be set depending on the context.

34.8.12 Structure IP_COAP_OPTIONS_INFO

Description

CoAP options description.

Prototype

```
typedef struct {
  U32                       OptionPresentMask;  // Mask of IP_COAP_OPTMASK_xxx.
  //
  U8*                       pHost;
  U8*                       pURI;
  U8*                       pQuery;
  U8*                       pETag;
  U8*                       pBlockLast;
  U8*                       pProxyURI;
  U8*                       pProxyScheme;
  IP_COAP_IF_MATCH_INFO*    pIfMatch;
  U32                       Observe;
  U32                       MaxAge;
  U32                       Size1;
  U32                       Block2;
  U32                       Block1;
  U32                       Size2;
  U16                       URIPort;
  U16                       Accept;
  U16                       ContentFormat;
  U16                       ProxyURILength;
  U8                        ProxySchemeLength;
  U8                        HostLength;
  U8                        URILength;
  U8                        QueryLength;
  U8                        ETagLength;
} IP_COAP_OPTIONS_INFO;
```

Member	Description
OptionPresentMask	Mask indicating the presence of options (IP_COAP_OPTMASK_xxx).
pHost	Pointer on the Uri-Host.
pURI	Pointer on the Uri-Path.
pQuery	Pointer on the Uri-Query.
pETag	Pointer on the ETag.
pBlockLast	Pointer used internally.
pProxyURI	Pointer on the Proxy-Uri.
pProxyScheme	Pointer on the Proxy-Scheme.
pIfMatch	If-Match linked list start.
Observe	Observe value.
MaxAge	Max-Age value.
Size1	Size1 value.
Block2	Block2 value.
Block1	Block1 value.
Size2	Size2 value.
URIPort	URIPort value.
Accept	Accept value.
ContentFormat	Content-Format value.

Member	Description
ProxyURILength	Length of pProxyURI.
ProxySchemeLength	Length of pProxyScheme.
HostLength	Length of pHost.
URILength	Length of pURI.
QueryLength	Length of pQuery.
ETagLength	Length of pETag.

Additional information

In the callbacks, it is recommended to access the options through the utility functions instead of directly accessing this structure as some options might have been reformatted.

34.8.13 Structure IP_COAP_IF_MATCH_INFO

Description

Utility structure to parametrize the If-Match option.

Prototype

```
struct {
  IP_COAP_IF_MATCH_INFO*   pNext;
  U8*                      pETag;
  U8                       ETagLength;
};
```

Member	Description
pNext	Anchor of the linked list.
pETag	Pointer on the ETag value.
ETagLength	Length of the ETag value.

Additional information

This list is used to allow requesting more than one ETag in the If-Match option. When the ETagLength is 0 (pETag is NULL), the check is for all ETags, meaning it is an “existence” verification.

34.8.14 Structure IP_COAP_HEADER_INFO

Description

Parsed CoAP header.

Prototype

```
typedef struct {
  U8  Type;
  U8  Code;
  U16 MessageId;
  U8  TokenLength;
  U8  aToken[8];
} IP_COAP_HEADER_INFO;
```

Member	Description
Type	CoAP message type (IP_COAP_TYPE_xxx).
Code	CoAP message code (IP_COAP_CODE_xxx).
MessageId	CoAP message id.
TokenLength	Length of aToken. Could be 0 if no token is present.
aToken	CoAP message token.

34.8.15 Structure IP_COAP_BLOCK_INFO

Description

Definition of the block used for Block1 and Block2 options.

Prototype

```
typedef struct {
  U16 Index;
  U16 Size;
} IP_COAP_BLOCK_INFO;
```

Member	Description
Index	Index of the block. 0 is the first block.
Size	Size of the block.

34.8.16 Structure IP_COAP_CONN_INFO

Description

UDP connection definition.

Prototype

```
typedef struct {
  void* hSock;
  union {
    U8  IPAddrV6[16];
    U32 IPAddrV4;
  };
  U16   Port;
  U8    Family;
} IP_COAP_CONN_INFO;
```

Member	Description
hSock	Socket information of the UDP connection. This field is not used by the CoAP processing, but only by the application callbacks pfReceive and pfSend. It could thus have any format adapted to the system.
IPAddrV6	IPv6 address.
IPAddrV4	IPv4 address
Port	UDP Port.
Family	Indication to use IPv4 (IP_COAP_IPV4) or IPv6 (IP_COAP_IPV6) address.

34.8.17 Callback pfReceive

Description

Callback used to receive UDP packets.

Prototype

```
int    (*pfReceive)(U8*                pBuffer,
                    unsigned           BufferSize,
                    IP_COAP_CONN_INFO* pInfo,
                    unsigned*          pIsMulticast);
```

Member	Description
pBuffer	Start of the message buffer.
BufferSize	Length of the message buffer.
pInfo	UDP connection information.
pIsMulticast	Flag to set to indicate that the received message is a multicast/broadcast (1) or a unicast (0) message. If not known always set to 0.

34.8.18 Callback pfSend

Description

Callback used to send UDP packets.

Prototype

```
int    (*pfSend)(U8* pBuffer, unsigned BufferSize, IP_COAP_CONN_INFO* pInfo);
```

Member	Description
pBuffer	Start of the message buffer.
BufferSize	Length of the message.
pInfo	UDP connection information.

34.8.19 Callback pfGetTimeMs

Description

Callback used to get the current time in milliseconds.

Prototype

```
U32    (*pfGetTimeMs)(void);
```


34.9 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the CoAP client/server presented in the tables below have been measured on a Cortex-M4 system with the default configuration.

34.9.1 Server ROM usage on a Cortex-M4 system

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

Addon	ROM
emNet CoAP server	approximately 9.2 kBytes

34.9.2 Client ROM usage on a Cortex-M4 system

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

Addon	ROM
emNet CoAP client	approximately 6.5 kBytes

34.9.3 Server RAM usage.

All of the RAM used by the server is taken from task stacks or from the application configured buffers. The application needs to provide:

- Server main context: A context of type `IP_COAP_SERVER_CONTEXT` needs 132 bytes.
- Message buffer: Minimum size is 272 bytes (blocks of 16 bytes), maximum 1500.
- Server resources: A resource of type `IP_COAP_SERVER_DATA` needs 48 bytes plus all the string defining the resource (Uri-Path) if in memory (could be constant though).
- Client context: At least one client context of type `IP_COAP_SERVER_CLIENT_INFO` with a size of 144 bytes is needed. Depending on the load of the server, more clients may be required.
- Observer context: It is needed only if server provides observable data. One observer context of type `IP_COAP_OBSERVER` needs 56 bytes.

For a minimal server configuration without observable resources: 548 + Number of resources x 48 bytes.

34.9.4 Client RAM usage.

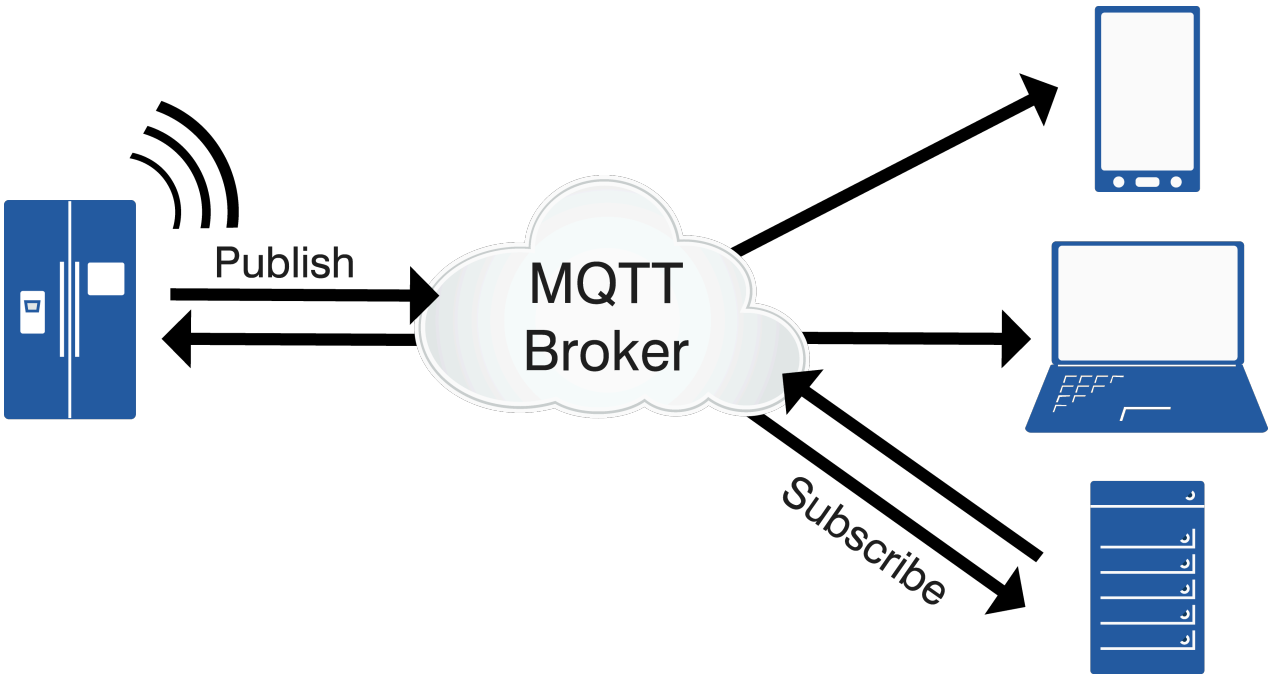
All of the RAM used by the client is taken from task stacks or from the application configured buffers. The application needs to provide:

- Client main context: A context of type `IP_COAP_CLIENT_CONTEXT` needs 184 bytes.
- Message buffer: Minimum size is 272 bytes (blocks of 16bytes), maximum 1500.
- Observer context: It is needed only when registering to an observable data. One observer context of type `IP_COAP_CLIENT_OBS` needs 32 bytes.

Chapter 35

MQTT client (Add-on)

The emMQTT client is an optional extension to emNet. The MQTT (Message Queuing Telemetry Transport) client can be used with emNet or with a different TCP/IP stack. All functions that are required to add the MQTT client to your application are described in this chapter.



35.1 emMQTT client

The emMQTT 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 MQTT client implements the relevant parts of the following Standard.

Standard#	Description
MQTT V 3.1.1 Plus Errata 01	MQTT Version 3.1.1 Plus Errata 01 *docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.pdf*
MQTT V 5	MQTT Version 5 *https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html*

The following table shows the contents of the emMQTT client root directory:

Directory	Content
`.\Application\`	Contains the example application to run the MQTT client with emNet.
`.\Config\`	Contains the MQTT client configuration file. Refer to *MQTT client configuration* on page for detailed information.
`.\IP\`	Contains the MQTT client sources and header files, `IP_MQTT_CLIENT.c` and `IP_MQTT_CLIENT.h`.
`.\Windows\IP\`	Contains the source, the project files and executables to run the emMQTT client on a Microsoft Windows host.

35.2 Feature list

- Full MQTT version 3.1.1 support.
- Full MQTT version 5 support.
- Publish/subscribe client included.
- Support for Quality of Service data delivery.
- 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.

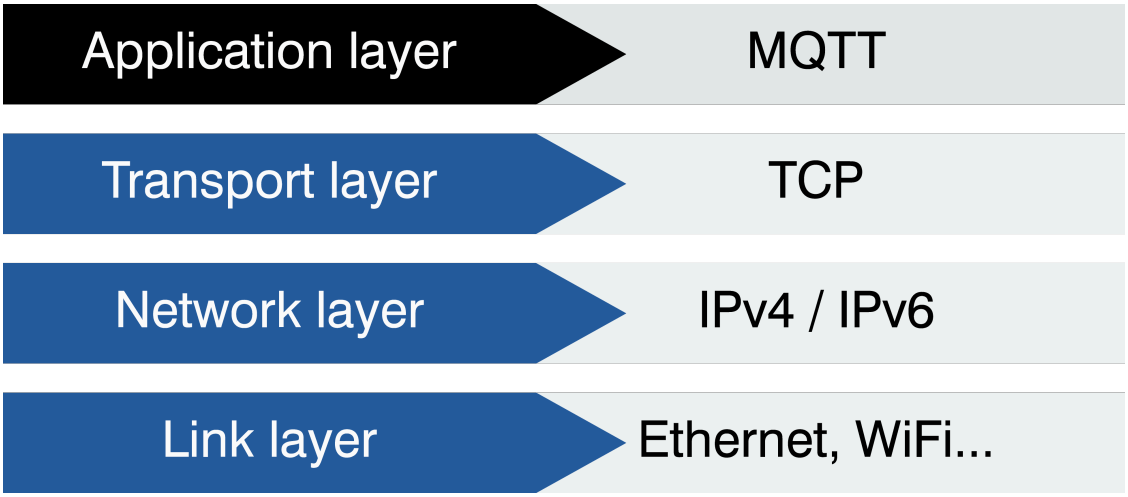
35.3 Requirements

TCP/IP stack

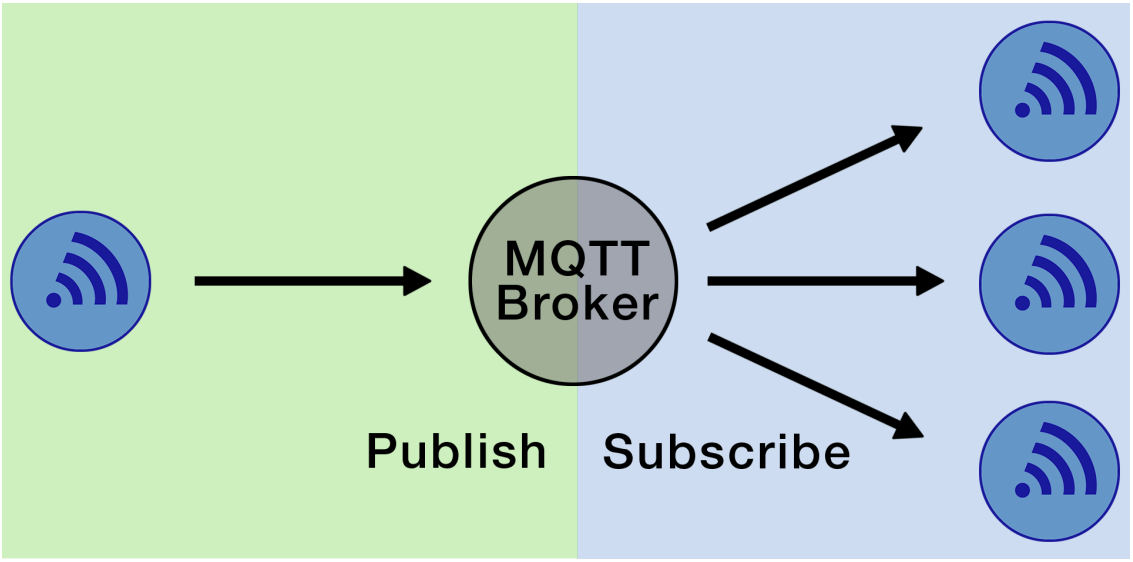
The emMQTT client requires a TCP/IP stack. It is optimized for emNet, but any RFC-compliant TCP/IP stack can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API and an implementation which uses the socket API of emNet.

35.4 MQTT backgrounds

MQTT is a very lightweight communication protocol originally designed for communication in M2M (Machine-To-Machine) contexts. It is easy to implement on the client side and has only a minimal packet overhead. This makes it ideal for the use with devices with limited resources.



It uses the publish/subscribe pattern, which is an alternative to the well known client/server model. In opposite to the client/server model, where a client directly communicates with an endpoint, the publish/subscribe pattern decouples the sender and receiver of a particular message. In the MQTT context the sending client is called publisher, the receiving client is called subscriber.



Publisher and subscriber do not know about the existence of one another. To enable the transport of a message, a third party is required. This third party is called a broker in the MQTT context. The broker distributes all incoming messages from the publishers to the subscribers, filtered on the topics the subscribers have assigned to.

MQTT uses topic-based filtering of messages. Publishers send topic related messages, subscribers receive messages if they have subscribed the topic.

To get messages a from a MQTT broker, a subscriber establishes a connection to the broker. The broker checks if a publisher has sent a message for the subscribed topic and if so, sends it to the subscriber. The advantage of this approach is that publisher and subscriber do not need to know each other and that they do not need to run at the same time. All they need to know is the IP address of the broker.

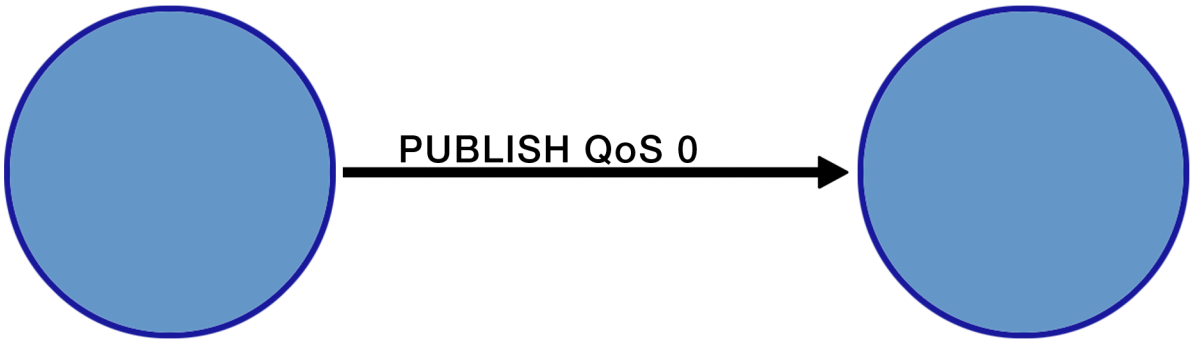
35.4.1 MQTT Quality of service

emMQTT client delivers application messages according to the chosen Quality of Service (QoS) level. The MQTT standard defines three qualities of service.

Quality of service	Description
QoS 0	QoS 0 is defined by the standard as “At most once”.
QoS 1	QoS 1 is defined by the standard as “At least once”.
QoS 2	QoS 2 is defined by the standard as “Exactly once”.

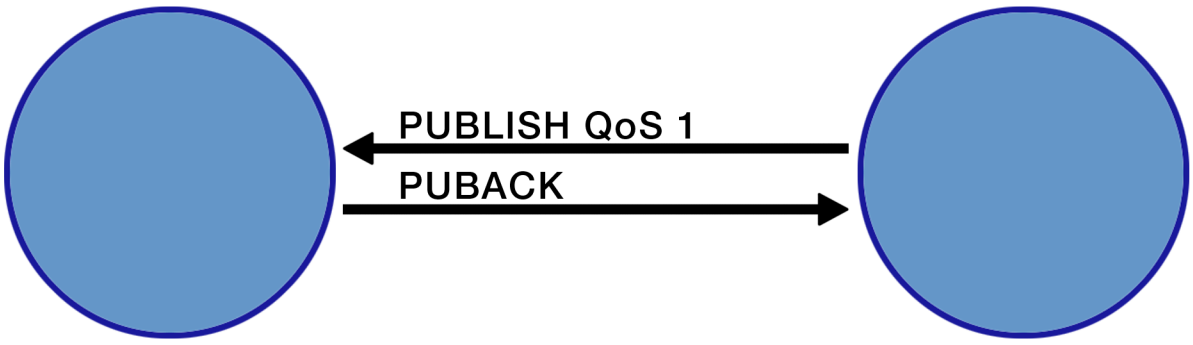
QoS 0: At most once delivery

Quality of service 0 means that the MQTT message is delivered according to the capabilities of the underlying network. The message arrives the receiver either once or not at all.



QoS 1: At least once delivery

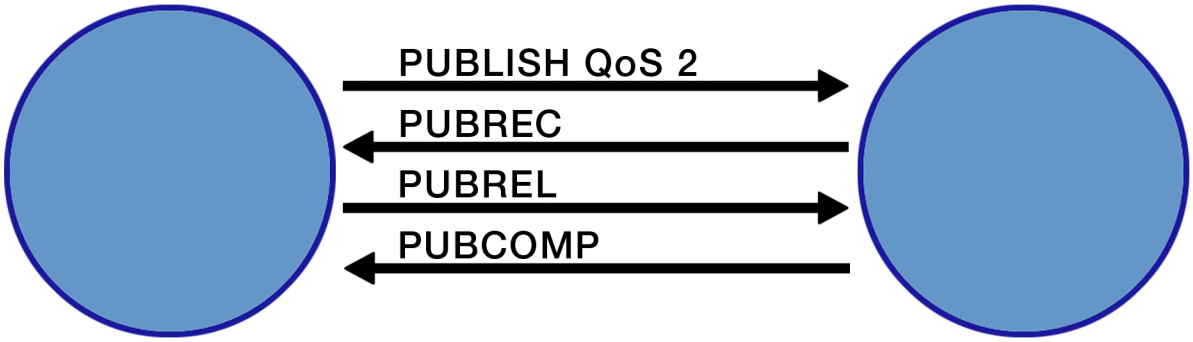
Quality of service 1 means that the MQTT message arrives at the receiver at least once. Each QoS 1 PUBLISH packet contains a packet identifier and is acknowledged by a PUBACK packet from the receiver.



The PUBACK packet includes the packet identifier of the PUBLISH message, which should be acknowledged.

QoS 2: Exactly once delivery

Quality of service 2 means that the MQTT message will be arrive exactly once. Each QoS 2 PUBLISH packet contains a packet identifier. The receiver acknowledges receipt with a two-step acknowledgment process.



The receipt of a PUBLISH message with QoS 2 is replied with a PUBREC message. The PUBREC packet includes the packet identifier of the PUBLISH message, which should be acknowledged. With the receipt of the PUBREC message knows the sender of the PUBLISH packet, that the message has been received and replies with a PUBREL packet. After the receiver gets the PUBREL packet it can discard every stored state and sends PUBCOMP.

QoS Downgrade

The quality of service is set between a single client and the broker and can be set differently for different topics. The QoS configured for the publisher and the subscriber are two different things.

In the following example client A is publishing to a topic with QoS2 and client B is subscribed to the same topic with QoS0. In this case the PUBLISH will arrive at the broker with QoS2 and the broker will perform a "QoS downgrade" before sending the PUBLISH to the client B. Client B will receive the message with QoS0.

QoS Upgrade

While the broker will downgrade QoS levels to meet subscriber's requirements the broker will never perform a QoS upgrade on incoming PUBLISH packets before sending them to topic subscribers.

Example:

- Client A sends PUBLISH with QoS1
- Client B subscribes with QoS 2 —> receives the message with QoS 1
- Client C subscribes with QoS 1 —> receives the message with QoS 1
- Client D subscribes with QoS 0 —> receives the message with QoS 0

35.5 emMQTT client configuration

The emMQTT 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).

Compile time configuration switches

Type	Symbolic name	Default	Description
F	`IP_MQTT_CLIENT_WARN`	--	Defines a function to output warnings. In debug configurations (`DEBUG = 1`) `IP_MQTT_CLIENT_WARN` maps to `IP_Warnf_Application()`.
F	`IP_MQTT_CLIENT_LOG`	--	Defines a function to output logging messages. In debug configurations (`DEBUG = 1`) `IP_MQTT_CLIENT_LOG` maps to `IP_Logf_Application()`.
B	`IP_MQTT_CLIENT_SUPPORT_V5`	1	Enables MQTT v5 support. Setting this to 0 will slightly reduce needed memory requirements.

35.6 API functions

Function	Description
MQTT client configuration functions	
IP_MQTT_CLIENT_Init()	Initializes an MQTT client.
IP_MQTT_CLIENT_SetLastWill()	Sets the last will message of an MQTT client.
IP_MQTT_CLIENT_SetUserPass()	Configures the username and password to be used to connect to a broker.
IP_MQTT_CLIENT_SetKeepAlive()	Sets the KeepAlive period to be given to the broker when connecting to it.
MQTT client operational functions	
IP_MQTT_CLIENT_ConnectEx()	Connects a client to an MQTT broker.
IP_MQTT_CLIENT_Disconnect()	Disconnects a client from an MQTT broker.
IP_MQTT_CLIENT_Publish()	Publishes a message.
IP_MQTT_CLIENT_Subscribe()	Subscribes to one or more topics with a broker.
IP_MQTT_CLIENT_Unsubscribe()	Unsubscribes from one or more topics registered with a broker.
IP_MQTT_CLIENT_WaitForNextMessage()	Waits for the next message ready to be received.
IP_MQTT_CLIENT_Recv()	Reads the payload of an MQTT message received.
IP_MQTT_CLIENT_Timer()	Timer function to send MQTT ping packets.
IP_MQTT_CLIENT_Exec()	Maintenance and house keeping.
IP_MQTT_CLIENT_ParsePublishEx()	Parses a given PUBLISH message.
IP_MQTT_CLIENT_IsClientConnected()	Checks if the client is connected to a broker.
MQTT debug functions	
IP_MQTT_Property2String()	Converts the Property ID into a string.
IP_MQTT_ReasonCode2String()	Converts the Reason Code into a string.
MQTT deprecated functions	
IP_MQTT_CLIENT_Connect()	Deprecated, use IP_MQTT_CLIENT_ConnectEx().
IP_MQTT_CLIENT_ParsePublish()	Deprecated please use IP_MQTT_CLIENT_ParsePublishEx().

35.6.1 IP_MQTT_CLIENT_Init()

Description

Initializes an MQTT client.

Prototype

```
int IP_MQTT_CLIENT_Init(      IP_MQTT_CLIENT_CONTEXT       * pClient,
                              char                         * pBuffer,
                              U32                            BufferSize,
                        const IP_MQTT_CLIENT_TRANSPORT_API * pAPI,
                        const IP_MQTT_CLIENT_APP_API       * pAppAPI,
                        const char                         * sId);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.
pBuffer	Pointer to a memory block, which is used by the client to process MQTT messages.
BufferSize	Buffer size of pBuffer.
pAPI	Pointer to IP_MQTT_CLIENT_TRANSPORT_API structure.
pAppAPI	Pointer to IP_MQTT_CLIENT_APP_API structure.
sId	Pointer to a string which holds the client ID. This ID must be unique for each client, normally an application should generate a unique client ID for each device.

Return value

≥ 0 O.K.
= -1 Parameter error.

Additional information

The application needs to make sure that this function does not get called while IP_MQTT_CLIENT_Timer() is active as it modifies internal lists that are accessed by it.

35.6.2 IP_MQTT_CLIENT_SetLastWill()

Description

Sets the last will message of an MQTT client.

Prototype

```
int IP_MQTT_CLIENT_SetLastWill(IP_MQTT_CLIENT_CONTEXT * pClient,
                               IP_MQTT_CLIENT_MESSAGE * pLastWill);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.
pLastWill	Pointer to IP_MQTT_CLIENT_MESSAGE structure which holds the message to set as last will on the broker.

Return value

0	O.K., last will set.
-1	Error, could not set last will (currently connected ?).

Additional information

This function needs to be called before connect as the last will parameter is given to the broker during connect. The pLastWill message needs to remain valid until IP_MQTT_CLIENT_ConnectEx() is done and for every further calls to IP_MQTT_CLIENT_ConnectEx().

35.6.3 IP_MQTT_CLIENT_SetUserPass()

Description

Configures the username and password to be used to connect to a broker.

Prototype

```
int IP_MQTT_CLIENT_SetUserPass(      IP_MQTT_CLIENT_CONTEXT * pClient,
                               const char                   * sUser,
                               const char                   * sPass);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.
sUser	Username to use.
sPass	Password to use.

Return value

0 O.K.
-1 Error, could not set user/pass (currently connected ?).

Additional information

This function is optional. Username and password are only required if the broker needs them for for authentication.

This function needs to be called before connect as the username and password is given to the broker during connect. sUser and sPass need to remain valid until IP_MQTT_CLIENT_ConnectEx() is done and for every further calls to IP_MQTT_CLIENT_ConnectEx().

35.6.4 IP_MQTT_CLIENT_SetKeepAlive()

Description

Sets the KeepAlive period to be given to the broker when connecting to it.

Prototype

```
int IP_MQTT_CLIENT_SetKeepAlive(IP_MQTT_CLIENT_CONTEXT * pClient,
                                U16                      KeepAlive);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.
KeepAlive	KeepAlive period [s] told to the broker.

Return value

0 O.K.
-1 Error, could not set timeout (currently connected ?).

Additional information

The KeepAlive is a time interval measured in seconds. Expressed as a 16-bit word, it is the maximum time interval that is permitted to elapse between the point at which the client finishes transmitting one control packet and the point it starts sending the next.

If the KeepAlive value is non-zero and the broker does not receive a control packet from the client within one and a half times the KeepAlive time period, it MUST disconnect the network connection to the client as if the network had failed.

This function is optional. By default the KeepAlive value is set to zero, since TCP layer can ensure the stability by using TCP KEEPALIVE packets. If you configure a non-zero value you need to call the MQTT timer function on a regular basis. For further information please refer to IP_MQTT_CLIENT_Timer().

35.6.5 IP_MQTT_CLIENT_ConnectEx()

Description

Connects a client to an MQTT broker. Establishes a TCP connection and handles the MQTT connect.

Prototype

```
int IP_MQTT_CLIENT_ConnectEx(      IP_MQTT_CLIENT_CONTEXT * pClient,
                             const IP_MQTT_CONNECT_PARAM  * pConnectPara,
                                   U8                     * pReasonCode);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.
pConnectPara	Pointer to IP_MQTT_CONNECT_PARAM structure.
pReasonCode	Pointer to an U8 to store the received Reason Code (MQTT 5 only). Can be NULL.

Return value

0	O.K.
-1	Error on transport layer.

Additional information

One IP_MQTT_CLIENT_CONTEXT supports only one connection at the same time, if you need to connect to multiple servers simultaneously you need to initialize a separate IP_MQTT_CLIENT_CONTEXT with IP_MQTT_CLIENT_Init().

35.6.6 IP_MQTT_CLIENT_Disconnect()

Description

Disconnects a client from an MQTT broker.

Prototype

```
int IP_MQTT_CLIENT_Disconnect(IP_MQTT_CLIENT_CONTEXT * pClient);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.

Return value

> 0 O.K., client disconnected.
= -1 Error, already disconnected ?

Additional information

Disconnecting means that the MQTT client, sends a MQTT disconnect packet and closes the transport layer connection.

35.6.7 IP_MQTT_CLIENT_Publish()

Description

Publishes a message.

Prototype

```
int IP_MQTT_CLIENT_Publish(IP_MQTT_CLIENT_CONTEXT * pClient,
                           IP_MQTT_CLIENT_MESSAGE * pPublish);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.
pPublish	Pointer to an IP_MQTT_CLIENT_MESSAGE structure.

Return value

> 0	Keep message.
= 0	O.K., PUBLISH message sent.
= -1	Error, could not send PUBLISH message.

Additional information

The IP_MQTT_CLIENT_MESSAGE structure should be allocated via the same alloc function which has been registered with IP_MQTT_CLIENT_Init() inside IP_MQTT_CLIENT_APP_API. If QoS 0 is used the user is responsible for freeing the IP_MQTT_CLIENT_MESSAGE structure. If QoS 1 is used the MQTT client will call the free callback when PUBACK is received. If QoS 2 is used the MQTT client will call the free callback when PUBCOMP is received. In case of an error (return -1) the user must free (or re-use) the IP_MQTT_CLIENT_MESSAGE structure.

35.6.8 IP_MQTT_CLIENT_Subscribe()

Description

Subscribes to one or more topics with a broker.

Prototype

```
int IP_MQTT_CLIENT_Subscribe(IP_MQTT_CLIENT_CONTEXT   * pClient,
                             IP_MQTT_CLIENT_SUBSCRIBE * pSubscribe);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.
pSubscribe	Pointer to IP_MQTT_CLIENT_SUBSCRIBE structure.

Return value

> 0 O.K.
= -1 Error, could not subscribe.

35.6.9 IP_MQTT_CLIENT_Unsubscribe()

Description

Unsubscribes from one or more topics registered with a broker.

Prototype

```
int IP_MQTT_CLIENT_Unsubscribe(IP_MQTT_CLIENT_CONTEXT   * pClient,
                               IP_MQTT_CLIENT_SUBSCRIBE * pUnsubscribe);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.
pUnsubscribe	Pointer to IP_MQTT_CLIENT_SUBSCRIBE structure.

Return value

0	O.K.
-1	Error, could not unsubscribe.

35.6.10 IP_MQTT_CLIENT_WaitForNextMessage()

Description

Waits for the next message ready to be received.

Prototype

```
int IP_MQTT_CLIENT_WaitForNextMessage(IP_MQTT_CLIENT_CONTEXT * pClient,
                                      U32                    * pType,
                                      U32                    * pNumBytesRecv,
                                      char                   * pBuffer,
                                      U32                      BufferSize);
```

Parameters

Parameter	Description
pClient	Pointer to an MQTT client structure.
pType	Pointer to an U32 to store the packet type.
pNumBytesRecv	Pointer to an U32 to store how much data can be received using IP_MQTT_CLIENT_Recv().
pBuffer	Pointer to the buffer to store the topic of the received message. Can be NULL.
BufferSize	Size of the buffer at pBuffer.

Return value

> 0	Length of the message topic received.
= 0	Connection has been gracefully closed by the broker.
< 0	Error.

35.6.11 IP_MQTT_CLIENT_Recv()

Description

Reads the payload of an MQTT message received.

Prototype

```
int IP_MQTT_CLIENT_Recv(IP_MQTT_CLIENT_CONTEXT * pClient,
                        char                   * pBuffer,
                        U32                      BufferSize);
```

Parameters

Parameter	Description
pClient	Pointer to an MQTT client structure.
pBuffer	Pointer to buffer to store data from message.
BufferSize	Size of buffer at pBuffer.

Return value

> 0 O.K., number of bytes received.
= 0 Connection has been gracefully closed by the broker.
< 0 Error.

Additional information

Can be called multiple times with a small buffer to read the payload in chunks.

35.6.12 IP_MQTT_CLIENT_Timer()

Description

Timer function to send MQTT ping packets.

Prototype

```
void IP_MQTT_CLIENT_Timer(void);
```

Additional information

In case `IP_MQTT_CLIENT_SetKeepAlive()` is used, the timer has to be called on a regular basis to satisfy your own configuration that has been set using `IP_MQTT_CLIENT_SetKeepAlive()`.

When KeepAlives are enabled MQTT brokers usually disconnect clients when they have not received a keepalive within 1.5 KeepAlive times.

35.6.13 IP_MQTT_CLIENT_Exec()

Description

Maintenance and house keeping.

Prototype

```
int IP_MQTT_CLIENT_Exec(IP_MQTT_CLIENT_CONTEXT * pClient);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.

Return value

> 0	O.K.
= 0	Connection gracefully closed by peer.
< 0	Error.

Additional information

This function is required, if the MQTT client will be called from several tasks. Please refer to the sample application IP_MQTT_CLIENT_PublisherSubscriber_2Tasks.c for detailed information.

35.6.14 IP_MQTT_CLIENT_ParsePublishEx()

Description

Parses a given PUBLISH message. The message must be completely available inside the given buffer. For QoS1 and QoS2 this function must be called to ensure that the PacketID of the message is set correctly in the IP_MQTT_CLIENT_MESSAGE structure for further QoS processing.

Prototype

```
int IP_MQTT_CLIENT_ParsePublishEx
                             (const IP_MQTT_CLIENT_CONTEXT  * pClient,
                                    IP_MQTT_CLIENT_MESSAGE  * pPublish,
                                    char                    * pBuffer,
                                    int                       NumBytes,
                                    char                   ** ppTopic,
                                    int                     * pNumBytesTopic,
                                    char                   ** ppPayload,
                                    int                     * pNumBytesPayload,
                                    char                   ** ppProperties,
                                    int                     * pNumBytesProperties);
```

Parameters

Parameter	Description
pClient	Pointer to IP_MQTT_CLIENT_CONTEXT structure.
pPublish	Pointer to a IP_MQTT_CLIENT_MESSAGE structure.
pBuffer	Pointer to the start of the publish message.
NumBytes	Number of bytes stored in the buffer.
ppTopic	Pointer to the topic of the publish message. Can be NULL.
pNumBytesTopic	Pointer to an int store the length of the topic. Can be NULL.
ppPayload	Pointer to the payload of the publish message. Can be NULL.
pNumBytesPayload	Pointer to an int to store the length of the payload. Can be NULL.
ppProperties	Pointer to the start of the properties. Can be NULL.
pNumBytesProperties	Pointer to an int to store the length of the properties. Can be NULL.

Return value

= 0 O.K.
< 0 Error.

35.6.15 IP_MQTT_CLIENT_IsClientConnected()

Description

Checks if the client is connected to a broker.

Prototype

```
int IP_MQTT_CLIENT_IsClientConnected(const IP_MQTT_CLIENT_CONTEXT * pClient);
```

Parameters

Parameter	Description
pClient	Pointer to an MQTT client structure.

Return value

> 0 Connected
= 0 Not connected

Additional information

This function is only required, if the MQTT client will be called from several tasks. Please refer to the sample application IP_MQTT_CLIENT_PublisherSubscriber_2Tasks.c for detailed information.

35.6.16 IP_MQTT_Property2String()

Description

Converts the Property ID into a string.

Prototype

```
char *IP_MQTT_Property2String(U8 x);
```

Parameters

Parameter	Description
x	Property ID to convert.

Return value

Pointer to a string which contains the property in text form.

35.6.17 IP_MQTT_ReasonCode2String()

Description

Converts the Reason Code into a string.

Prototype

```
char *IP_MQTT_ReasonCode2String(U8 x);
```

Parameters

Parameter	Description
x	Reason Code to convert.

Return value

Pointer to a string which contains the Reason Code in text form.

35.6.18 IP_MQTT_CLIENT_Connect()

Description

Deprecated, use `IP_MQTT_CLIENT_ConnectEx()`. Connects a client to an MQTT broker. Establishes a TCP connection and handles the MQTT connect.

Prototype

```
int IP_MQTT_CLIENT_Connect(      IP_MQTT_CLIENT_CONTEXT * pClient,
                           const char                   * sAddr,
                                 U16                      Port,
                                 U8                       CleanSession);
```

Parameters

Parameter	Description
`pClient`	Pointer to `IP_MQTT_CLIENT_CONTEXT` structure.
`sAddr`	String with address of the broker.
`Port`	Listening port of the broker.
`CleanSession`	• 0: Reuse old session data to continue. • 1: Start with a clean session.

Return value

0	O.K., client connected.
-1	Error.

Additional information

One `IP_MQTT_CLIENT_CONTEXT` supports only one connection at the same time, if you need to connect to multiple servers simultaneously you need to initialize a separate `IP_MQTT_CLIENT_CONTEXT` with `IP_MQTT_CLIENT_Init()`.

35.6.19 IP_MQTT_CLIENT_ParsePublish()

Description

Deprecated please use `IP_MQTT_CLIENT_ParsePublishEx()`. Parses a given PUBLISH message. The message must be completely available inside the given buffer.

Prototype

```
int IP_MQTT_CLIENT_ParsePublish(IP_MQTT_CLIENT_MESSAGE  * pPublish,
                                char                    * pBuffer,
                                int                       NumBytes,
                                char                   ** ppTopic,
                                int                     * pNumBytesTopic,
                                char                   ** ppPayload,
                                int                     * pNumBytesPayload);
```

Parameters

Parameter	Description
pPublish	Pointer to a `IP_MQTT_CLIENT_MESSAGE` structure.
pBuffer	Pointer to the start of the publish message.
NumBytes	Number of bytes stored in the buffer.
ppTopic	Pointer to the topic of the publish message. Can be `NULL`.
pNumBytesTopic	Pointer to an int store the length of the topic. Can be `NULL`.
ppPayload	Pointer to the payload of the publish message.
pNumBytesPayload	Pointer to an int to store the length of the payload.

Return value

= 0 O.K.
< 0 Error.

35.7 Data structures

Structure	Description
IP_MQTT_CLIENT_TRANSPORT_API	Transport API function pointer.
IP_MQTT_CLIENT_MESSAGE	Message maintenance structure.
IP_MQTT_CLIENT_TOPIC_FILTER	Structure used to subscribe to a particular topic.
IP_MQTT_CLIENT_SUBSCRIBE	Structure used by the IP_MQTT_CLIENT_Subscribe() function to subscribe to many topics at once.
IP_MQTT_CONNECT_PARAM	Structure containing parameters required for a new connection to a MQTT broker.
IP_MQTT_PROPERTY	Structure describing a MQTT 5 Property.
IP_MQTT_CONNECT_PARAM	Structure containing parameters required for a new connection to a MQTT broker.
IP_MQTT_STR_PAIR_DATA	Structure describing a MQTT 5 String Pair Property.
IP_MQTT_STR_DATA	Structure describing a MQTT 5 String Property.
IP_MQTT_BIN_DATA	Structure describing a MQTT 5 Binary Property.

35.7.1 IP_MQTT_CLIENT_TRANSPORT_API

Description

Structure with pointers to the required socket interface functions.

Prototype

```
typedef struct IP_MQTT_CLIENT_TRANSPORT_API {
  void*    (*pfConnect)   (      char*    SrvAddr,
                                 unsigned SrvPort );
  void     (*pfDisconnect)(      void*    pSocket );
  int      (*pfReceive)   (      void*    pSocket
  char*    pData,
  int      Len );
  int      (*pfSend)      (      void*    pSocket
  const char*    pData,
  int      Len );
} IP_MQTT_CLIENT_TRANSPORT_API;
```

Member	Description
pfConnect	Pointer to the connect function (for example, `connect()`).
pfDisconnect	Pointer to the disconnect function (for example, `closesocket()`).
pfSend	Pointer to a callback function (for example, `send()`).
pfRecv	Pointer to a callback function (for example, `recv()`).

35.7.2 IP_MQTT_CLIENT_APP_API

Description

Structure with pointers to the required functions.

Prototype

```
typedef struct IP_MQTT_CLIENT_APP_API {
  U16   (*pfGenRandom)           ( void );
  void* (*pfAlloc)               ( U32   NumBytesReq );
  void  (*pfFree)                ( void* p );
  void  (*pfLock)                ( void );
  void  (*pfUnlock)              ( void );
  int   (*pfRecvMessage)         ( void* pMQTTClient,
                                   void* pPublish,
                                   int   NumBytesRem );
  // Get the subscribed message
  int   (*pfOnMessageConfirm)    ( void* pMQTTClient,
                                   U8    Type,
                                   U16   PacketId );
  // Inform the application that a message has been processed
  int   (*pfHandleError)         ( void* pMQTTClient );
  // Inform the application that the publishing of a message failed.
  void  (*pfHandleDisconnect)    ( void* pMQTTClient,
                                   U8 ReasonCode);
  int   (*pfOnMessageConfirmEx)  ( void* pMQTTClient,
                                   U8 Type,
                                   U16 PacketId,
                                   U8 ReasonCode);
  int   (*pfRecvMessageEx)       ( void* pMQTTClient,
                                   void* pPublish,
                                   int NumBytesRem,
                                   U8 * pReasonCode);
  void  (*pfOnProperty)          ( void* pMQTTClient,
                                   U16 PacketId,
                                   U8 PacketType,
                                   IP_MQTT_PROPERTY * pProp);
} IP_MQTT_CLIENT_APP_API;
```

Member	Description
pfGenRandom	Pointer to a function which returns a random unsigned short value. The random value is used for the packet ID of MQTT packets.
pfAlloc	Pointer to the Alloc() function. This function is only required, if you plan to call the MQTT client API from several tasks (e.g. MQTT client acts as subscriber and publisher.)
pfFree	Pointer to the Free () function. This function is only required, if you plan to call the MQTT client API from several tasks (e.g. MQTT client acts as subscriber and publisher.)
pfLock	Pointer to a function which acquires a lock to ensure MQTT API is thread safe. This function is only required, if you plan to call the MQTT client API from several tasks (e.g. MQTT client acts as subscriber and publisher.)
pfUnlock	Pointer to a function which releases a previously acquired lock for thread safety MQTT API. This function is only required, if you plan to call the MQTT client API from several tasks (e.g. MQTT client acts as subscriber and publisher.)
pfRecvMessage	Deprecated, use pfRecvMessageEx().
pfOnMessage-Confirm	Deprecated, use pfOnMessageConfirmEx().

Member	Description
pfHandleError	Pointer to a callback which is called in case of an error.
pfHandleDis-connect	[MQTT 5 only] Pointer to an optional callback which is called when the server sends a Disconnect Request. Can be NULL.
pfOnMessage-ConfirmEx	Pointer to a callback which is called when all QoS related messages are processed for a message. Sent messages can be freed after this callback has been called. Received messages can be processed by the application after this callback has been called.
pfRecvMes-sageEx	Pointer to a callback which is called when a PUBLISH message is received.
pfOnProperty	[MQTT 5 only] Pointer to an optional callback which is called when a non-PUBLISH (CONNACK, PUBACK, PUBREC, PUBCOMP, SUBACK, UNSUBACK, DISCONNECT, AUTH) message is received with a Property. The callback is called for each contained property individually. Can be NULL.

35.7.3 IP_MQTT_CLIENT_MESSAGE

Description

Message maintenance structure.

Type definition

```
typedef struct {
  IP_MQTT_DLIST_ITEM         Link;
  const char               * sTopic;
  const char               * pData;
  U32                        DataLen;
  U16                        PacketId;
  U8                         QoS;
  U8                         Retain;
  U8                         Duplicate;
  const IP_MQTT_PROPERTY **  paProperties;
  U8                         NumProperties;
} IP_MQTT_CLIENT_MESSAGE;
```

Structure members

Member	Description
Link	Internal link.
sTopic	Pointer to the topic string.
pData	Pointer to the payload which should be published.
DataLen	Number of payload bytes.
PacketId	Packet ID. (Will be filled by the MQTT client module)
QoS	QoS flag.
Retain	Retain flag.
Duplicate	Duplicate flag. (Will be filled by the MQTT client module)
paProperties	[MQTT 5 only] Array containing pointers to IP_MQTT_PROPERTY structures. Can be NULL.
NumProperties	[MQTT 5 only] Number of elements inside paProperties.

35.7.4 IP_MQTT_CLIENT_TOPIC_FILTER

Description

Structure used to subscribe to a particular topic.

Type definition

```
typedef struct {
  const char * sTopicFilter;
  U16          Length;
  U8           QoS;
} IP_MQTT_CLIENT_TOPIC_FILTER;
```

Structure members

Member	Description
sTopicFilter	Pointer to the topic filter string (zero-terminated).
Length	This field is deprecated! It may be removed in future versions. Make sure sTopicFilter is zero-terminated.
QoS	Quality of service flag.

35.7.5 IP_MQTT_CLIENT_SUBSCRIBE

Description

Structure used by the IP_MQTT_CLIENT_Subscribe() function to subscribe to many topics at once. The topics are described in an array of IP_MQTT_CLIENT_TOPIC_FILTER.

Type definition

```
typedef struct {
  IP_MQTT_DLIST_ITEM            Link;
  IP_MQTT_CLIENT_TOPIC_FILTER * pTopicFilter;
  int                           TopicCnt;
  U16                           PacketId;
  U8                            ReturnCode;
  U8                          * paReasonCodes;
} IP_MQTT_CLIENT_SUBSCRIBE;
```

Structure members

Member	Description
Link	Internal link.
pTopicFilter	Pointer to the first topic filter structure.
TopicCnt	Number of added topics.
PacketId	Packet ID. (Will be filled by the MQTT client module.)
ReturnCode	Return code. (Will be filled by the MQTT client module.)
paReasonCodes	[MQTT 5 only] Pointer to an array to store the received Reason Codes. Can be NULL. In case the application subscribes to multiple topics at once this buffer will be filled with the reason codes for each topic. Reason codes will appear in the same order as the subscribed topics, e.g. For a subscription with two topics t1 (QoS1) and t2 (QoS2) you will receive IP_MQTT_REASON_GRANTED_QOS_1, IP_MQTT_REASON_GRANTED_QOS_2 in the buffer. If the pointer is not NULL the size of this buffer MUST be the same as TopicCnt. When using pfRecvMessageEx care must be taken because the buffer is filled with valid values only after the pfRecvMessageEx callback is called with PacketType SUBACK and the correct Packet ID.

35.7.6 IP_MQTT_PROPERTY

Description

Structure describing a MQTT 5 Property.

Prototype

```
typedef struct _IP_MQTT_PROPERTY {
  IP_MQTT_CLIENT_PROP_TYPE PropType;            // Type of the property.
  union {
    U8  Data_U8;                                // Byte storage.
    U16 Data_U16;                               // Half-word storage.
    U32 Data_U32;                               // Word storage.
    // Variable integers are stored by the user in a U32,
    // the MQTT module will do the encoding in the Variable Integer format.
    U32 Data_VarInt;
    // Structure of type IP_MQTT_BIN_DATA.
    struct IP_MQTT_BIN_DATA Data_Bin;
    // Structure of type IP_MQTT_STR_DATA.
    struct IP_MQTT_STR_DATA Data_Str;
    // Structure of type IP_MQTT_STR_PAIR_DATA.
    struct IP_MQTT_STR_PAIR_DATA Data_StrPair;
  } PropData;
} IP_MQTT_PROPERTY;
```


35.7.7 IP_MQTT_CONNECT_PARAM

Description

Structure containing parameters required for a new connection to a MQTT broker.

Type definition

```
typedef struct {
  const char               * sAddr;
  U16                        Port;
  U8                         CleanSession;
  U8                         Version;
  const IP_MQTT_PROPERTY **  paProperties;
  U8                         NumProperties;
} IP_MQTT_CONNECT_PARAM;
```

Structure members

Member	Description
sAddr	String with address of the broker.
Port	Listening port of the broker.
CleanSession	• 0: Reuse old session data to continue. • 1: Start with a clean session.
Version	Following values are allowed: • 4 - use MQTT 3.1.1 • 5 - use MQTT 5
paProperties	[MQTT 5 only] Array containing pointers to IP_MQTT_PROPERTY structures. Can be NULL.
NumProperties	[MQTT 5 only] Number of elements inside paProperties.

35.7.8 IP_MQTT_STR_PAIR_DATA

Description

Structure describing a MQTT 5 String Pair Property. This is exclusivly used by `IP_MQTT_PROP_TYPE_USER_PROPERTY`.

Type definition

```
typedef struct {
  U16          Len1;
  U16          Len2;
  const char * pData1;
  const char * pData2;
} IP_MQTT_STR_PAIR_DATA;
```

Structure members

Member	Description
Len1	Length of the first string.
Len2	Length of the second string.
pData1	Pointer to a buffer containing the first string.
pData2	Pointer to a buffer containing the second string.

35.7.9 IP_MQTT_STR_DATA

Description

Structure describing a MQTT 5 String Property. Strings should not be terminated with .

Type definition

```
typedef struct {
  U16          Len;
  const char * pData;
} IP_MQTT_STR_DATA;
```

Structure members

Member	Description
Len	Length of the string.
pData	Pointer to a buffer containing the string.

35.7.10 IP_MQTT_BIN_DATA

Description

Structure describing a MQTT 5 Binary Property.

Type definition

```
typedef struct {
  U16        Len;
  const U8 * pData;
} IP_MQTT_BIN_DATA;
```

Structure members

Member	Description
Len	Number of bytes inside the buffer pointed to by pData.
pData	Pointer to a buffer containing the data.

35.8 Resource usage

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

35.8.1 Resource usage on a Cortex-M4 system

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

35.8.1.1 ROM usage

Addon	ROM
emMQTT client	approximately 2.5 kBytes

35.8.1.2 RAM usage

Addon	RAM
emMQTT client context w/o task stack	approximately 60 Bytes

Chapter 36

WebSocket (Add-on)

The emNet WebSocket IoT (Internet of Things) protocol is an optional extension to emNet. The WebSocket add-on can be used with emNet or with a different TCP/IP stack that uses a socket API. All functions that are required to add WebSocket support to your application are described in this chapter.

36.1 emNet WebSocket support

The emNet WebSocket add-on is an optional extension which adds WebSocket functionality to your application. It has been implemented with the limitations of an embedded system in mind while still providing a flexible but powerful API paired with a small memory footprint. The RAM usage has been kept to a minimum by smart handling of the protocols characteristics.

The WebSocket add-on implements the relevant parts of the following Request For Comments (RFC).

RFC#	Description
[RFC 6455]	The WebSocket Protocol Direct download: *ftp://ftp.rfc-editor.org/in-notes/rfc6455.txt*

The following table shows the contents of the emNet WebSocket package root directory:

Directory	Content
`.\Application\`	Contains the example applications to run a WebSocket printf() server/client with emNet.
`.\IP\`	Contains the WebSocket sources `IP_WEBSOCKET.c` and `IP_WEBSOCKET.h`.
`.\Windows\IP\WebSocket_printf_Server\`	Contains the source, the project files and an executable to run the emNet WebSocket add-on (server side) on a Microsoft Windows host. Refer to *Using the WebSocket samples* on page 1188 for detailed information.
`.\Windows\IP\WebSocket_printf_Client\`	Contains the source, the project files and an executable to run the emNet WebSocket add-on (client side) on a Microsoft Windows host. Refer to *Using the WebSocket samples* on page 1188 for detailed information.

36.2 Feature list

- Converts a synchronous HTTP connection into an asynchronous data connection.
- Can traverse firewalls using the well known HTTP port.
- Can be used with or without web server.
- Seamless integration with the emNet web server add-on.
- Low memory footprint.
- Optimized API for embedded systems.
- Independent of the TCP/IP stack: any stack with sockets can be used.
- Project for executable on PC for Microsoft Visual Studio included.

36.3 Requirements

TCP/IP stack

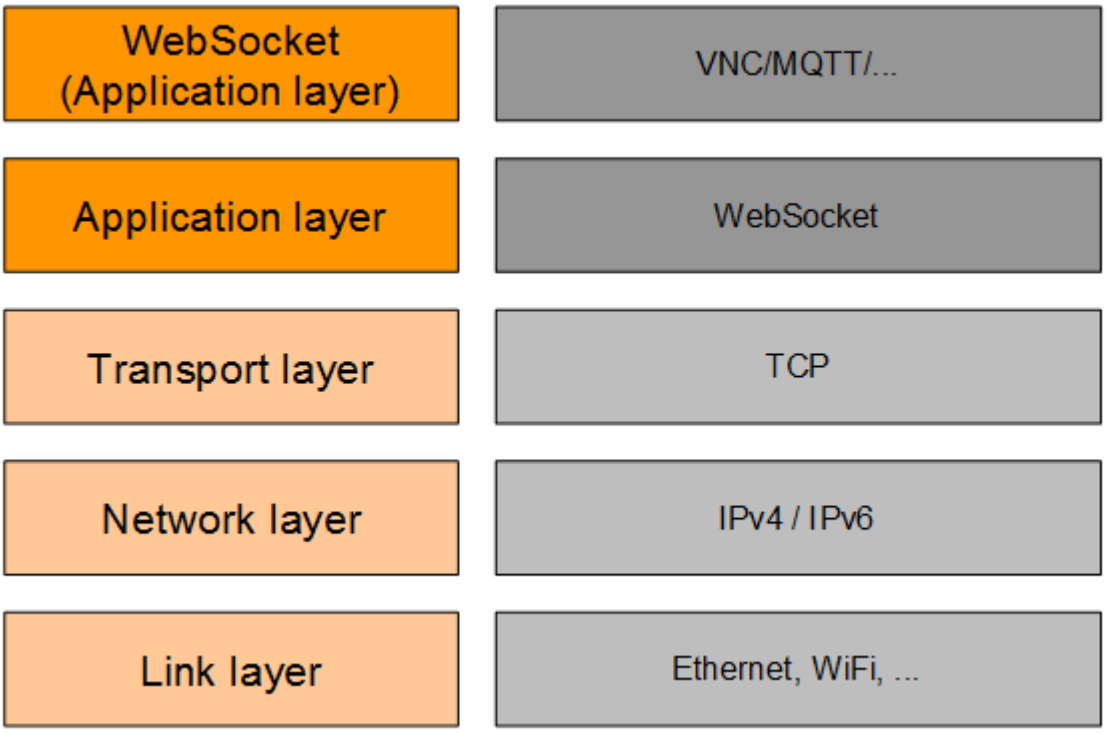
The emNet WebSocket add-on requires a TCP/IP stack. It is optimized for emNet, but any RFC-compliant TCP/IP stack that has a socket API can be used. The shipment includes a Win32 simulation, which uses the standard Winsock API as well as an implementation which uses the socket API of emNet.

36.4 Backgrounds

The WebSocket protocol is an `IoT` (`Internet of Things`) protocol that allows to upgrade a regular synchronous HTTP connection into an asynchronous bidirectional data tunnel. It allows to establish a regular HTTP connection with a webserver, checking if both sides, client and server, understand the WebSocket protocol depending on the answer sent back by the webserver.

The non intrusive test for checking the WebSocket protocol being supported is compatible with the HTTP/1.1 standard and will return a page with an error code in worst case if the protocol is not supported by the webserver. The webserver service for serving regular pages remains fully operational while coexisting with the WebSocket protocol.

The WebSocket protocol is using the TCP/IP protocol for data, using the HTTP/1.1 protocol only to initially establish a WebSocket connection.



An advantage of the WebSocket protocol is that it establishes a connection through the HTTP protocol. This allows an easier handling in terms of firewalls as typically the standard port TCP 80 is allowed for outgoing connections, allowing WebSocket clients to work without problems. For a WebSocket server, offering webpages and WebSocket tunneled protocols only a single port is required to be opened in the firewall or allowed to be forwarded by a router from the Internet to your local network.

36.4.1 Establishing a WebSocket connection

Before application data can be transferred, a connection handshake needs to be done. The handshake is done sending a regular HTTP/1.1 protocol request to a webserver that contains WebSocket specific extension fields in the HTTP header as well as a list of WebSocket subprotocols that the client is able to use.

```
GET /printf HTTP/1.1
Host: 192.168.11.158
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: ew+1nlIjRf4c+zVtzqIZ2g==
Sec-WebSocket-Protocol: debug
```

The sample above shows the minimum HTTP header that is sent by a WebSocket client to a webserver to establish a WebSocket connection. The following header fields are used in the request:

Header field	Explanation
Request	The very first line expected contains the METHOD used for access (for WebSocket this is always “GET”), the resource that is accessed (in this sample “/printf”) and the HTTP protocol version used (“HTTP/1.1”).
Host	The host name to access. For webservers that host multiple domains the “Host:” field specifies the domain to use.
Connection	Extra actions requested for the HTTP connection. In the sample above a connection “Upgrade” is requested. The type of upgrade is then specified by the “Upgrade:” field.
Upgrade	The type of connection upgrade requested. In the sample above the connection shall be upgraded to the “websocket” protocol.
Sec-WebSocket-Version	The highest WebSocket protocol version supported by the client. 13 is the first final WebSocket protocol version.
Sec-WebSocket-Key	Key value that is used to verify that the webserver is really understanding the WebSocket protocol.
Sec-WebSocket-Protocol	WebSocket subprotocols supported by the client. In the sample above we use the sample subprotocol “debug”.

36.4.2 Accepting a WebSocket connection

To accept a WebSocket connection the webserver needs to send back a HTTP header to the client confirming the protocol upgrade. In case the webserver is not aware of the WebSocket protocol or the resource requested is not available, the webserver can send back its regular HTTP error code and content.

The client expects a HTTP 101 “Switching Protocols” response in case the webserver supports the WebSocket protocol. Any other HTTP code returned by the webserver means that the WebSocket protocol, the resource or the selected subprotocol is not supported.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Protocol: debug
Sec-WebSocket-Accept: fwgzotwyvIXhoZ/o77F3O8wc8Pc=
```

The sample above is the minimum HTTP response header sent back by a webserver to accept the WebSocket connection upgrade. The following header fields are different in the response:

Header field	Explanation
Response	The response code “101” confirms that a protocol switch has been done. The explanation “Switching Protocols” is typically added but should only serve as additional text that might differ.
Sec-WebSocket-Protocol	WebSocket subprotocols supported by the client. In the sample above we use the sample subprotocol “debug”.
Sec-WebSocket-Accept	This value is calculated based on the value of the `Sec-WebSocket-Key` field from the request and is an additional indicator that the WebSocket protocol is fully understood by the webserver.

36.4.3 Closing a WebSocket connection

Closing a WebSocket connection is basically very easy. As the underlying protocol is TCP, a close of the TCP socket means closing the WebSocket connection as well. While this will always work it is suggested to first close the WebSocket connection itself before closing the TCP socket. This allows sending the other side a reason why the connection is closed. This can be done using the function `IP_WEBSOCKET_Close()`.

36.4.4 WebSocket data framing

The WebSocket protocol combines the following properties of TCP and UDP:

TCP properties

- Lost data is retransmitted.
- Data is received in the correct order.
- The connection status can be checked by using TCP KEEPALIVES.

UDP properties

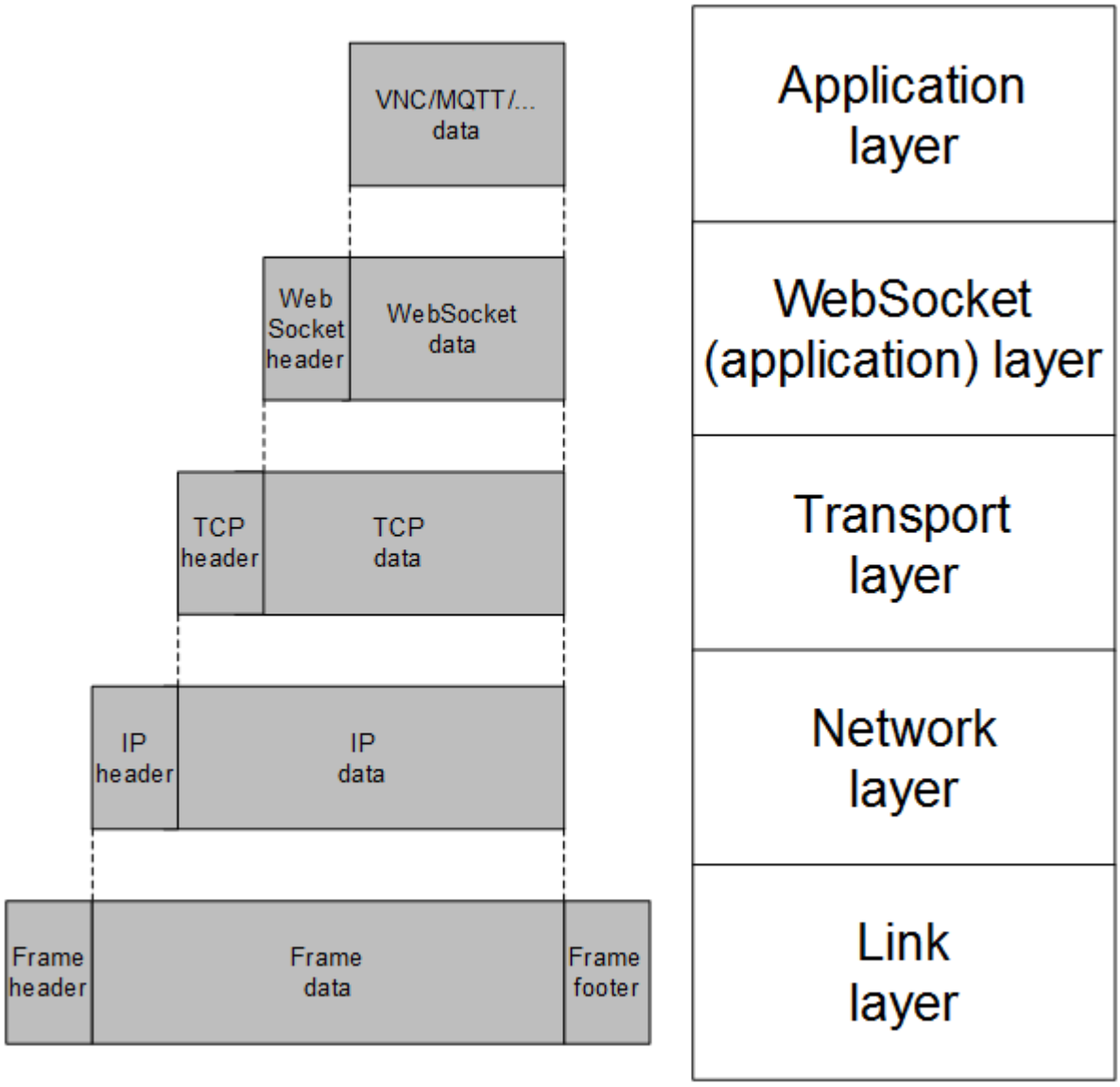
- Data can be sent in small frames (while being part of a larger message).
- The length of the frame is known (the complete message length is not).
- Control frames (UDP: other data) can be sent between data frames.

Principle of operation

Application data sent using the WebSocket protocol can be described as UDP like framing of data within a TCP data stream. While this might sound complex at first, the protocol itself is not.

The main purpose of the WebSocket protocol is to allow a device with limited resources sending one big block of data in chunks (called a frames in the WebSocket protocol) whenever data is ready/generated instead of collecting the data before sending. At the same time control commands should be able to be processed anytime between frames. At the same time the TCP protocol ensures that data is not lost and frames will be sent and received in the correct order.

The WebSocket protocol achieves these goals by sending all data in frames that have a header upfront. The encapsulation of the protocol is shown below:



The WebSocket header size is between 2 and 14 bytes and contains information about the position of the frame in context to the complete data to send (first, one of the middle, last frame of a message) as well as the length of the application data in this single frame.

36.4.5 WebSocket frame types

WebSocket frames contain a frame type in their header. The following types are available for WebSocket protocol version 13:

Frame type	ID	Description
IP_WEBSOCKET_FRAME_TYPE_CONTINUE	0x00	The data in this frame is continuation of the last frame with an ID other than 0x00.
IP_WEBSOCKET_FRAME_TYPE_TEXT	0x01	Used for sending an unterminated textual string.
IP_WEBSOCKET_FRAME_TYPE_BINARY	0x02	Used to send any type of data.
IP_WEBSOCKET_FRAME_TYPE_CLOSE	0x08	Used to close the message with a close code and an optional message.
IP_WEBSOCKET_FRAME_TYPE_PING	0x09	PING request.
IP_WEBSOCKET_FRAME_TYPE_PONG	0x0A	PING response.

Typically there are only two frame types that are relevant for the application. These are IP_WEBSOCKET_FRAME_TYPE_TEXT and IP_WEBSOCKET_FRAME_TYPE_BINARY. They specify the type of a received message. However if the application uses a custom protocol and does not need to distinguish between text and binary data the frame type IP_WEBSOCKET_FRAME_TYPE_BINARY can always be used.

All other frame types are control frame types. Control frames can be sent in between message frames to allow an immediate processing instead of having to wait for all frames of a message to be transferred. Controls frames are handled by the WebSocket protocol in the background. The only exception to this is a frame of the type IP_WEBSOCKET_FRAME_TYPE_CLOSE as this allows the application to close the WebSocket connection gracefully sending a close code itself and an optional message.

36.5 Using the WebSocket samples

Ready to use examples for Microsoft Windows and emNet are supplied. If you use another TCP/IP stack, the samples have to be adapted.

The supplied sample application `IP_WEBSOCKET_printf_Server.c` and `GUI_VNC_X_StartServer.c` show different approaches for implementing the WebSocket protocol. While both accept a WebSocket connection their implementation and handling of the connection is completely different.

36.5.1 IP_WEBSOCKET_printf_Server.c

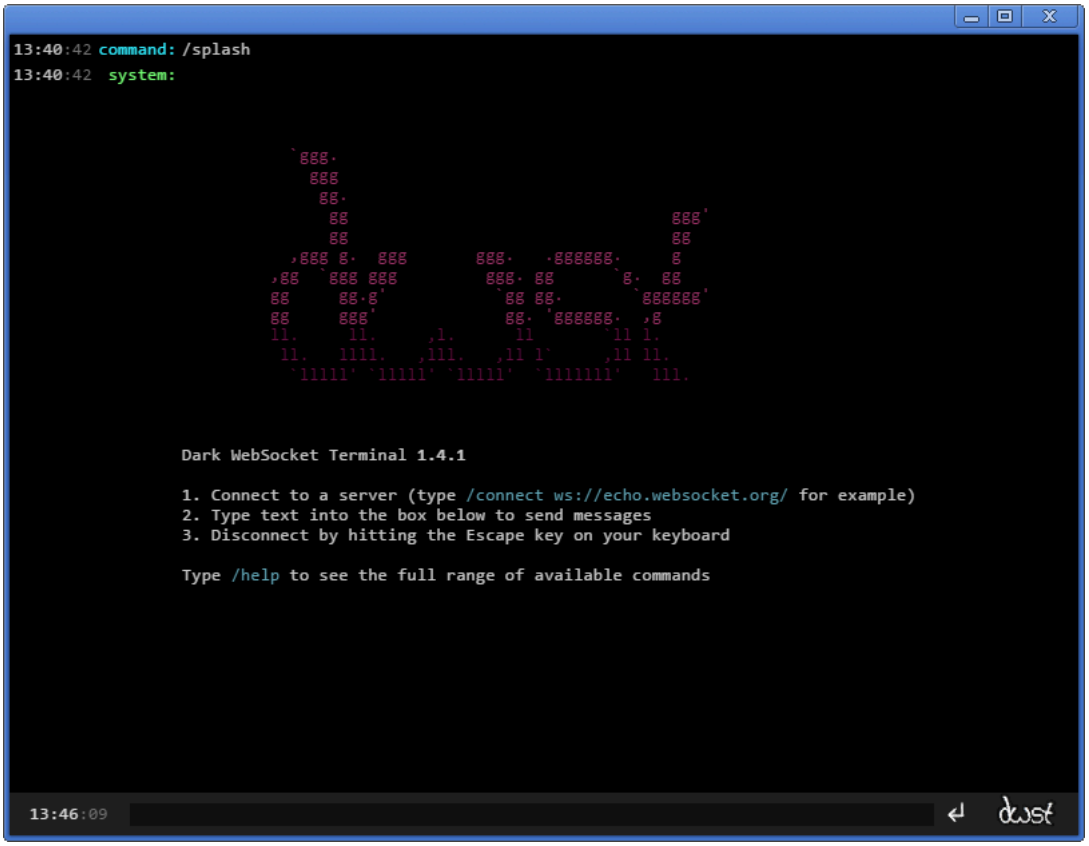
This sample accepts a WebSocket connection and awaits messages of the frame type `IP_WEBSOCKET_FRAME_TYPE_TEXT`. It tries to copy received text messages into a local buffer to string terminate them and then outputs them to the standard output using printf().

The WebSocket connection handshake is done without a web server by parsing the request HTTP header for the "Sec-WebSocket-Key:" header field value and sending back its counterpart in a HTTP response.

The message handling of the established WebSocket connection is handled by the same task that accepts the connection, blocking it for accepting further WebSocket connections while one connection is ongoing.

Testing the sample

To test the sample, a WebSocket client is necessary. One of these clients is the "Dark WebSocket terminal" (DWST) that is available as extension for the Chrome browser.



Connect

To connect to the WebSocket printf() server enter the following command:

```
/connect ws://<IP addr. of server>/printf debug
```


This will open a connection to the server using the resource "/printf" and suggesting the subprotocol "debug".

Send a text that the target will printf()

Once the WebSocket connection is established, text can be send to the server that will the be printf() processed. A "Hello World!" can be send using the following command:

```
/send Hello World!
```

Some characters need to be escaped using a backslash. In the above sample the whitespace needs to be written as "\ " (backslash followed by a space) as the whitespace is a separator character in case commands can have parameters.

Disconnect

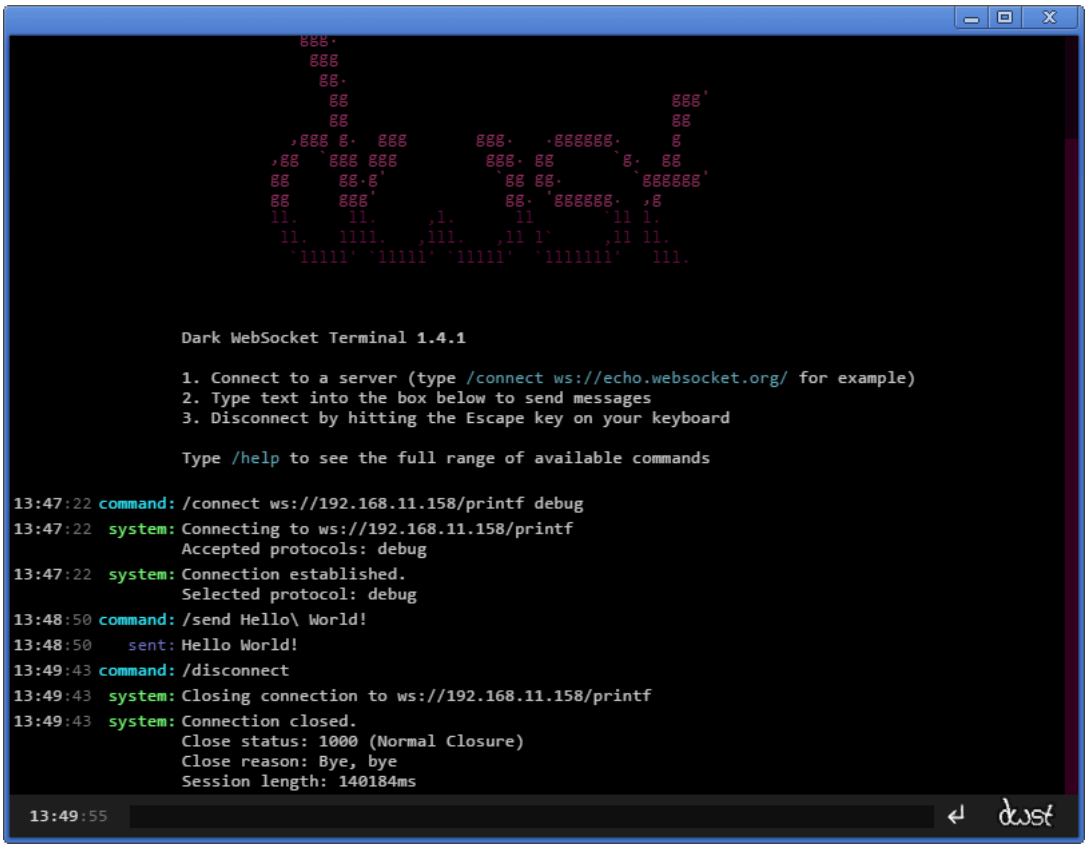
To disconnect from the server the following line can be executed:

```
/disconnect
```

This will send a frame of type `IP_WEBSOCKET_FRAME_TYPE_CLOSE` to the server. The server should then simply close the TCP connection. However in a good manner the server might send its own frame of type `IP_WEBSOCKET_FRAME_TYPE_CLOSE` back for a graceful close of the WebSocket connection before closing the TCP connection.

Sample output

The following is a screenshot of the commands described above:



The output of a terminal on the server side should show the following lines:

```
WebSocket client connected.
Client: Hello World!
WebSocket client disconnected.
```


36.5.2 GUI_VNC_X_StartServer.c

This sample consists of a part of the emWin VNC Server setup. The file `GUI_VNC_X_StartServer.c` contains the TCP/IP abstraction between emWin and a TCP/IP stack. It can be found in the folder "\Shared\GUI\Sample\". The sample adds a resource hook to the web server that allows using the `emWin VNC` server with the `noVNC` client, written entirely in HTML5.

The latest version of the `noVNC` client can be downloaded from the following location: https://github.com/kanaka/noVNC/releases

`noVNC` works best when used with the Chrome browser. Inconsistent HTML5 implementations in different browsers result in problems running `noVNC`.

Precautions

The following precautions need to be met for connecting with `noVNC`:

- The emNet web server add-on has been added to a target running the `emWin VNC` server.
- The define `GUI_VNC_SUPPORT_WEBSOCKET_SERVER` has been set to "1" in the project settings or the file itself to enable the WebSocket implementation.
- The WebSocket resource hooked into the web server is configured for the path "/websockify" and the subprotocol "binary". Both values are the defaults for `noVNC`.

By registering the WebSocket resource with the web server using `IP_WEBS_WEBSOCKET_AddHook()`, the web server remains able to serve its regular webpages while at the same time allowing to accept WebSocket connections under the registered resource path "/websockify" and executing the WebSocket handshake with a client.

An accepted WebSocket connection can be handled by the web server in two ways:

- Executing the WebSocket message handling from the same task context, blocking the web server resource (web server child task) from being reused until the WebSocket connection is closed.
- Dispatching the connection handle to another context, freeing the web server resource for another request.

For this sample the latter is used, dispatching the connection handle used by the web server to the VNC server task for further processing. Dispatching another WebSocket connection is disabled and further WebSocket connections will be discarded until the `noVNC` client gets disconnected.

Testing the sample

To test the noVNC client with the emWin VNC server with WebSocket support please make sure the precautions described above are met. Start the noVNC client by opening the file "vnc.html" from the noVNC package in your browser. The only thing to do is to enter the IP address or host name of your WebSocket/VNC server and the port the web server listens on as shown in the screenshot below:

By pressing the "Connect" button noVNC establishes a WebSocket connection with the web server. The established connection is then dispatched to the VNC server task as described before.

While it is possible to connect to the VNC server with a regular VNC client and noVNC, it is not possible at the same time due to the limitations of the sample.

36.5.3 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 emNet WebSocket add-on. If you do not have the Microsoft compiler, a precompiled executable of the "WebSocket printf Server" is also supplied.

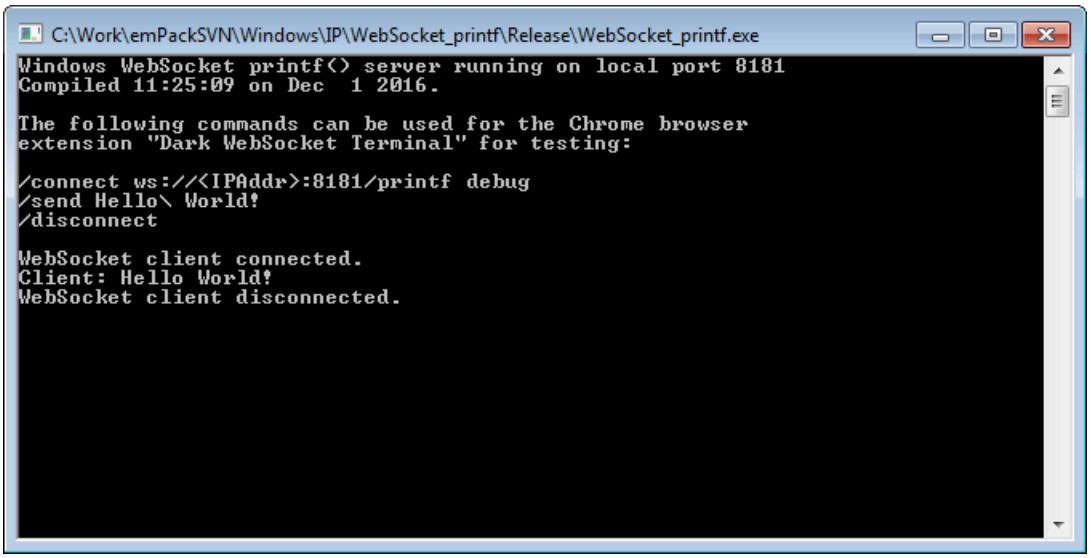
Building the sample program

Open the workspace `WebSocket_printf.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.

Testing the sample program

The server uses the IP address of the host PC on which it runs. Unlike the emNet sample, the Windows sample does not use port 80 but instead uses port 8181. This is due to port 80 being used by Windows by default from Windows 8 onwards.

The sample can be tested the same way as its emNet counterpart using a WebSocket client like Dark WebSocket Terminal (DWST). The sample shows some tests commands as online help during startup as shown below.



36.6 Configuration

The emNet WebSocket 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.

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).

Compile time configuration switches

Type	Symbolic name	Default	Description
N	`IP_WEBSOCKET_MAX_DATA_SIZE`	16384	Maximum amount of bytes that can be transferred in one WebSocket frame. The payload length of a WebSocket frame is an U64 value, however underlying transport layers like the BSD socket API typically do not support a payload that big. For the moment, data chunks that exceed this limit result in returning an error.

36.7 API functions

Function	Description
IP_WEBSOCKET_Close()	Sends a WebSocket CLOSE frame to the peer.
IP_WEBSOCKET_DiscardMessage()	Discards all frames that belong to the currently processed message.
IP_WEBSOCKET_GenerateAcceptKey()	Generates the value to send back for the Sec-WebSocket-Accept field when accepting the connection.
IP_WEBSOCKET_InitClient()	Initializes a client side WebSocket context.
IP_WEBSOCKET_InitServer()	Initializes a server side WebSocket context.
IP_WEBSOCKET_Recv()	Receives application data until the end (FIN flagged) frame of a message has been used up (all data read).
IP_WEBSOCKET_Send()	Sends application data.
IP_WEBSOCKET_WaitForNextMessage()	Waits until the next message is received.

36.7.1 IP_WEBSOCKET_Close()

Description

Sends a WebSocket CLOSE frame to the peer.

Prototype

```
int IP_WEBSOCKET_Close(IP_WEBSOCKET_CONTEXT * pContext,
                       char                 * sReason,
                       U16                    CloseCode);
```

Parameters

Parameter	Description
pContext	WebSocket connection context.
sReason	Reason for the close. Can be NULL. • For a client context this needs to point to RAM as the data will be encoded at its source location before it is sent.
CloseCode	IP_WEBSOCKET_CLOSE_CODE_* .

Return value

> 0	All data sent.
= 0	No data sent (typically with non-blocking sockets). Can indicate an unexpected/early close from the peer. Do the same call again. If it really was a close from peer a second call should return with an error. This return value depends on how the TCP/IP stack treats this case.
= IP_WEBSOCKET_ERR_AGAIN	No data (payload) sent, do the same call again.
< 0	Error.

Additional information

Return values 0 and IP_WEBSOCKET_ERR_AGAIN are typically only returned when the socket interface is used in non-blocking mode.

A close message is the last message received from a peer. It is unlikely and even wrong if the peer would send us any message after a close message.

Once a close message has been received from the peer, a close message shall be sent if not done before and then the network connection shall be closed.

For a server side the WebSocket close message shall be sent and then the network connection shall be closed without waiting for a close message from the peer.

36.7.2 IP_WEBSOCKET_DiscardMessage()

Description

Discards all frames that belong to the currently processed message.

Prototype

```
int IP_WEBSOCKET_DiscardMessage(IP_WEBSOCKET_CONTEXT * pContext);
```

Parameters

Parameter	Description
pContext	WebSocket connection context.

Return value

> 0	O.K., message discarded.
= 0	Connection closed.
< 0	Error.

Additional information

There are two types of WebSocket messages that can be received by the application (text and binary). Only a text or binary message that has been previously identified by `IP_WEBSOCKET_WaitForNextMessage()` will be discarded.

After discarding a message, the next message needs to be identified by calling `IP_WEBSOCKET_WaitForNextMessage()`.

36.7.3 IP_WEBSOCKET_GenerateAcceptKey()

Description

Generates the value to send back for the Sec-WebSocket-Accept field when accepting the connection.

Prototype

```
int IP_WEBSOCKET_GenerateAcceptKey(void * pSecWebSocketKey,
                                   int    SecWebSocketKeyLen,
                                   void * pBuffer,
                                   int    BufferSize);
```

Parameters

Parameter	Description
pSecWebSocketKey	Pointer to a buffer containing the string of the Sec-WebSocket-Key from the HTTP request.
SecWebSocketKeyLen	Number of characters of the Sec-WebSocket-Key (without string termination).
pBuffer	Buffer where to store the accept key.
BufferSize	Size of buffer where to store the accept key.

Return value

> 0 Length of accept key.
= 0 Error, buffer not big enough.

Additional information

The calculation of the accept key is done by the following steps:

- Adding the Base64 encoded Sec-WebSocket-Key to a new SHA-1 hash.

- Adding the string “258EAFA5-E914-47DA-95CA-C5AB0DC85B11” to the SHA-1 hash.

- Base64 encode the resulting SHA-1 hash. This is the accept key to send back with the Sec-WebSocket-Accept field.

36.7.4 IP_WEBSOCKET_InitClient()

Description

Initializes a client side WebSocket context.

Prototype

```
void IP_WEBSOCKET_InitClient(      IP_WEBSOCKET_CONTEXT       * pContext,
                             const IP_WEBSOCKET_TRANSPORT_API * pAPI,
                                   IP_WEBSOCKET_CONNECTION    * pConnection);
```

Parameters

Parameter	Description
pContext	WebSocket connection context.
pAPI	Application API to use for WebSocket communication.
pConnection	Transport connection handle.

36.7.5 IP_WEBSOCKET_InitServer()

Description

Initializes a server side WebSocket context.

Prototype

```
void IP_WEBSOCKET_InitServer(      IP_WEBSOCKET_CONTEXT       * pContext,
                             const IP_WEBSOCKET_TRANSPORT_API * pAPI,
                                   IP_WEBSOCKET_CONNECTION    * pConnection);
```

Parameters

Parameter	Description
pContext	WebSocket connection context.
pAPI	Application API to use for WebSocket communication.
pConnection	Transport connection handle.

36.7.6 IP_WEBSOCKET_Recv()

Description

Receives application data until the end (FIN flagged) frame of a message has been used up (all data read).

Prototype

```
int IP_WEBSOCKET_Recv(IP_WEBSOCKET_CONTEXT * pContext,
                      void                 * pData,
                      int                    NumBytes);
```

Parameters

Parameter	Description
pContext	WebSocket connection context.
pData	Where to store the received data.
NumBytes	Maximum amount of data to receive.

Return value

> 0 Amount of data received.
= 0 Connection closed.
< 0 Error. IP_WEBSOCKET_ERR_AGAIN : Repeat the call (typically with non-blocking sockets). IP_WEBSOCKET_ERR_ALL_DATA_READ: All data of the current message has been read. Call IP_WEBSOCKET_WaitForNextMessage() to check for another message.

Additional information

The WebSocket receive function follows the concept of a blocking socket recv call. Once called it will block until either at least one byte of application data from the current message is available or the peer of the WebSocket connection sends a CLOSE frame or an error occurs.

This function is able to receive continuous data from multiple WebSocket frames of the same message. In case this function is called with no more data left in the current frame and the frame was the last of the current message (FIN flag set), the error code IP_WEBSOCKET_ERR_ALL_DATA_READ is returned. In this case IP_WEBSOCKET_WaitForNextMessage() needs to be called before further recv calls.

Application data for a CLOSE frame (IP_WEBSOCKET_FRAME_TYPE_CLOSE) always starts with a U16 (network/big endian) CloseCode if application data is present at all. If more than these 2 bytes are available to be read as application data, the rest of the data is of an unknown type but is typically a textual reason for the close and can therefore be treated like receiving a TEXT frame of type IP_WEBSOCKET_FRAME_TYPE_TEXT .

36.7.7 IP_WEBSOCKET_Send()

Description

Sends application data.

Prototype

```
int IP_WEBSOCKET_Send(IP_WEBSOCKET_CONTEXT * pContext,
                      void                 * pData,
                      int                    NumBytes,
                      U8                     MessageType,
                      U8                     SendMore);
```

Parameters

Parameter	Description
pContext	WebSocket connection context.
pData	Data to send. • For a client context this needs to point to RAM as the data will be encoded at its source location before it is sent.
NumBytes	Amount of data to send.
MessageType	IP_WEBSOCKET_FRAME_TYPE_TEXT or IP_WEBSOCKET_FRAME_TYPE_BINARY .
SendMore	• 0: Send a single frame/part message or the last frame/part of a message sent in multiple calls. • 1: Send a message in multiple frames/parts. This is not the last frame/part of the message.

Return value

= NumBytes	All data sent.
= 0	No data sent (typically with non-blocking sockets).
= IP_WEBSOCKET_ERR_AGAIN	No data (payload) sent, do the same call again.
< 0	Error.

Additional information

Return values 0 and IP_WEBSOCKET_ERR_AGAIN are typically only returned when the socket interface is used in non-blocking mode.

36.7.8 IP_WEBSOCKET_WaitForNextMessage()

Description

Waits until the next message is received.

Prototype

```
int IP_WEBSOCKET_WaitForNextMessage(IP_WEBSOCKET_CONTEXT * pContext,
                                    U8                   * pMessageType);
```

Parameters

Parameter	Description
pContext	WebSocket connection context.
pMessageType	Where to store the read message type. Can be NULL.

Return value

> 0	O.K., next message received.
= 0	Connection closed.
= IP_WEBSOCKET_ERR_AGAIN	Repeat the call (typically with non-blocking sockets).
< 0	Error.

Additional information

The return value IP_WEBSOCKET_ERR_AGAIN is typically only returned when the socket interface is used in non-blocking mode.

O.K. is returned for all message types received but the application has to handle only what it is interested in (text, binary or close messages).

All other message types that are effective control frames are handled internal. Messages of type text or binary that are not handled by the application will be discarded on the next call of this function.

36.8 Data structures

36.8.1 Structure IP_WEBSOCKET_TRANSPORT_API

Description

Used to provide an interface to external TCP/IP transport functions.

Prototype

```
typedef struct IP_WEBSOCKET_TRANSPORT_API_STRUCT IP_WEBSOCKET_TRANSPORT_API;

struct IP_WEBSOCKET_TRANSPORT_API_STRUCT {
  int (*pfRecv)( IP_WEBSOCKET_CONTEXT*    pContext,
                 IP_WEBSOCKET_CONNECTION* pConnection,
                 void*                    pData,
                 unsigned                 NumBytes);
  int (*pfSend)(       IP_WEBSOCKET_CONTEXT* pContext,
                       IP_WEBSOCKET_CONNECTION* pConnection,
                 const void* pData,
                       unsigned NumBytes);
  U32 (*pfGenMaskKey)(void);
};
```

Member	Description
pfRecv	Callback used to receive data via a TCP/IP stack.
- pContext	WebSocket connection context.
- pConnection	Connection handle of the TCP/IP stack. Typically a socket handle or SSL session handle.
- pData	Where to store the received data.
- NumBytes	Maximum amount of data to receive.
pfSend	Callback used to send data via a TCP/IP stack.
- pContext	WebSocket connection context.
- pConnection	Connection handle of the TCP/IP stack. Typically a socket handle is used.
- pData	Data to send.
- NumBytes	Amount of data to send.

36.9 Resource usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the WebSocket add-on 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.

36.9.1 ROM usage on a Cortex-M4 system

The following resource usage has been measured on a Cortex-M4 system using SEGGER Embedded Studio V3.10e, size optimization.

Addon	ROM
emNet WebSocket protocol	approximately 1.7 kBytes

36.9.2 RAM usage

While the WebSocket add-on itself does not require any RAM at all, a context element per connection is required.

Addon	RAM
emNet WebSocket context	approximately 30 Bytes

Chapter 37

Profiling with SystemView

This chapter describes how to configure and enable profiling of emNet using SystemView.

37.1 Profiling overview

emNet is instrumented to generate profiling information of API functions and driver-level functions.

These profiling information expose the run-time behavior of emNet 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 emNet profiling instrumentation can be easily configured and set up.

37.2 Additional files for profiling

Additional files are required on target and PC side for full functionality of SystemView.

37.2.1 Additional files on target side

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.

37.2.2 Additional files on PC side

For fully functional and readable outputs in the SystemView PC application, a description file for the corresponding middleware is required. This description file extends the values sent from the target to fully readable text outputs.

While SystemView already comes with the most recent description files at the time the SystemView release has been built, these files might not be the latest available. The latest SystemView description files can be found in the emNet shipment in the folder `/Shared/SystemView/Description/`. You can copy these files over to the `Description` folder that comes with the SystemView package.

The version at the end of the SystemView description file does not have to match the exact version of the middleware it is used with. They are valid from this version onwards until a description file for a newer version is required.

37.3 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 emNet 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.

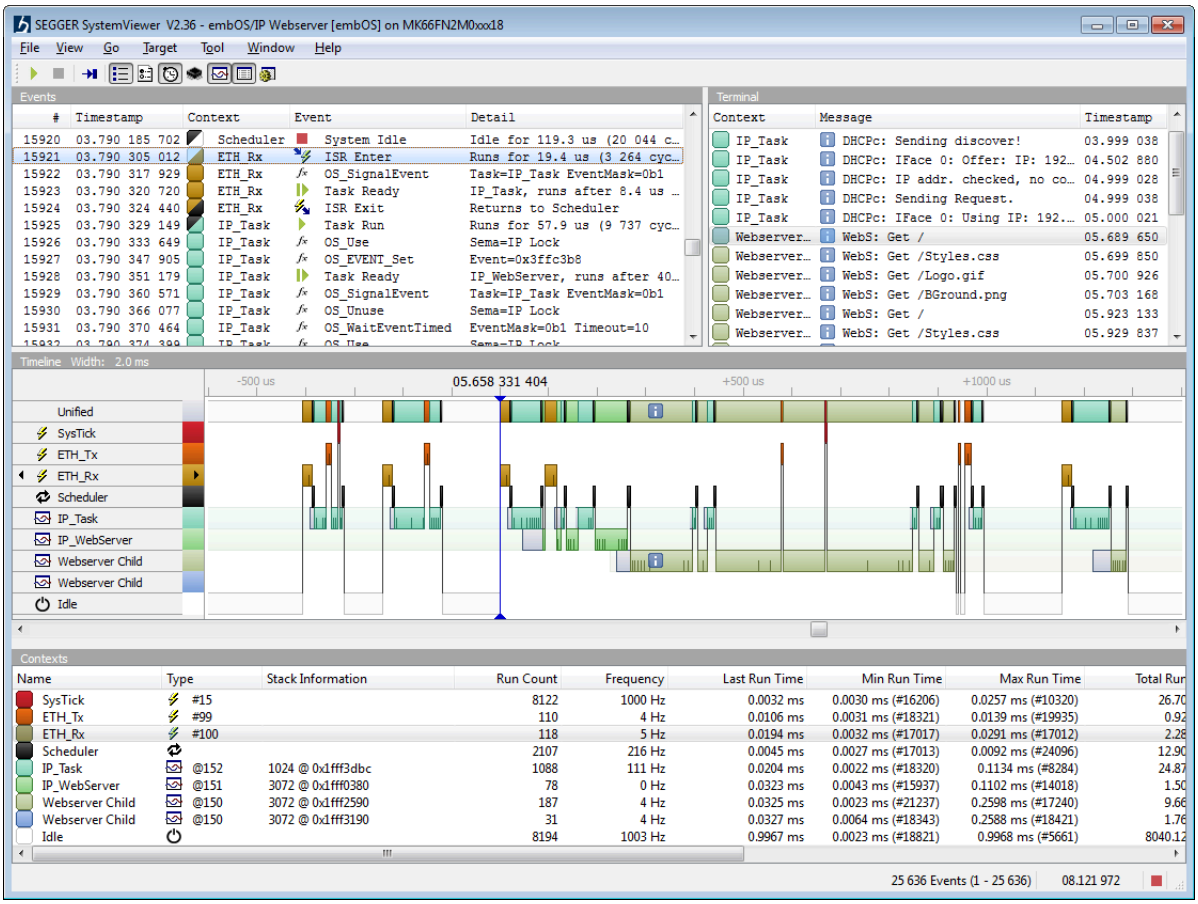
```
#if defined(SUPPORT_PROFILE) && (SUPPORT_PROFILE)
  #ifndef   IP_SUPPORT_PROFILE
    #define IP_SUPPORT_PROFILE           1
  #endif
#endif

#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
  ...
}
```


37.4 Recording and analyzing profiling information

When profiling is included and enabled emNet generates profiling events. On a system which supports RTT (i.e. ARM Cortex-M and Renesas RX) the data can be read and analyzed 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 postmortem mode. Please refer to the SystemView User Manual for more information.



Chapter 38

Debugging

emNet 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.

38.1 Message output

The debug builds of emNet 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 emNet sample configuration files. All logging messages are disabled except for the messages from the initialization and the DHCP setup phase.

38.2 Testing stability

emNet allows to define drop-rates for both receiver an transmitter. This feature can be used to simulate packet loss. Packet loss means that one or more packets fail to reach their destination. Packet loss can be caused by a number of factors (for example, signal degradation over the network medium, faulty networking hardware, error in network applications, etc.).

Two variables, `IP_TxDropRate` and `IP_RxDropRate`, are implemented to define the drop-rate while the target is running. There is no need to recompile the stack. The default value of these variables is 0, which means that no packets should be dropped from the stack. Any other value of n (for example, n = 2,3, …) will drop every n-th packet. This allows testing the reliability of communication and performance drop. A good value to test the stability is typically around 50.

To change the value of `IP_TxDropRate` and/or `IP_RxDropRate` the following steps are required:

1. Download your emNet 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.

38.3 API functions

Function	Description
IP_Log()	This function is called by the stack in debug builds with log output.
IP_Warn()	This function is called by the stack in debug builds with warning output.
IP_Logf_Application()	Outputs a formatted log message of type IP_MTYPE_APPLICATION if the filter allows it.
IP_Warnf_Application()	Outputs a formatted warn message of type IP_MTYPE_APPLICATION if the filter allows it.
Filter functions	
IP_AddLogFilter()	Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.
IP_AddWarnFilter()	Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.
IP_SetLogFilter()	Sets a mask that defines which logging message should be logged.
IP_SetWarnFilter()	Sets a mask that defines which warning messages should be logged.
General debug functions/macros	
IP_PANIC()	Called if the stack encounters a critical situation.
IP_Panic()	This function is called if the stack encounters a critical situation.

38.3.1 IP_Log()

Description

This function is called by the stack in debug builds with log output. In a release build, this function may not be linked in.

Prototype

```
void IP_Log(const char * s);
```

Parameters

Parameter	Description
s	String to output.

Additional information

Interrupts and task switches printf() has a re-entrance problem on a lot of systems if interrupts are not disabled. Strings to output would be scrambled if during an output from a task an output from an interrupt would take place. In order to avoid this problem, interrupts are disabled.

38.3.2 IP_Warn()

Description

This function is called by the stack in debug builds with warning output. In a release build, this function may not be linked in.

Prototype

```
void IP_Warn(const char * s);
```

Parameters

Parameter	Description
s	String to output.

Additional information

Interrupts and task switches printf() has a re-entrance problem on a lot of systems if interrupts are not disabled. Strings to output would be scrambled if during an output from a task an output from an interrupt would take place. In order to avoid this problem, interrupts are disabled.

38.3.3 IP_Logf_Application()

Description

Outputs a formatted log message of type `IP_MTYPE_APPLICATION` if the filter allows it.

Prototype

```
void IP_Logf_Application(const char * sFormat,
                                     ...);
```

Parameters

Parameter	Description
`sFormat`	String to output that might contain placeholders.

38.3.4 IP_Warnf_Application()

Description

Outputs a formatted warn message of type `IP_MTYPE_APPLICATION` if the filter allows it.

Prototype

```
void IP_Warnf_Application(const char * sFormat,
                                       ...);
```

Parameters

Parameter	Description
`sFormat`	String to output that might contain placeholders.

38.3.5 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 TypeMask);
```

Parameters

Parameter	Description
TypeMask	Bitwise-OR'ed message type(s) like IP_MTYPE_INIT.

Example

```
IP_AddLogFilter(IP_MTYPE_DRIVER); // Activate driver logging messages
```


38.3.6 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 TypeMask);
```

Parameters

Parameter	Description
TypeMask	Bitwise-OR'd message type(s) like IP_MTYPE_INIT.

Example

```
IP_AddWarnFilter(IP_MTYPE_DRIVER); // Activate driver warning messages
```


38.3.7 IP_SetLogFilter()

Description

Sets a mask that defines which logging message should be logged. Logging messages are only available in debug builds of the stack.

Prototype

```
void IP_SetLogFilter(U32 TypeMask);
```

Parameters

Parameter	Description
TypeMask	Bitwise-OR'd message type(s) like IP_MTYPE_INIT.

Additional information

This function should be called from IP_X_Config(). By default, the filter condition IP_MTYPE_INIT, IP_MTYPE_DHCP and IP_MTYPE_LINK_CHANGE are set.

38.3.8 IP_SetWarnFilter()

Description

Sets a mask that defines which warning messages should be logged. Warning messages are only available in debug builds of the stack.

Prototype

```
void IP_SetWarnFilter(U32 TypeMask);
```

Parameters

Parameter	Description
TypeMask	Bitwise-OR'd message type(s) like IP_MTYPE_INIT.

Additional information

This function should be called from IP_X_Config(). By default, all filter conditions are set.

38.3.9 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.

38.3.10 IP_Panic()

Description

This function is called if the stack encounters a critical situation. In a release build, this function may not be linked in.

Prototype

```
void IP_Panic(const char * s);
```

Parameters

Parameter	Description
s	String to output.

38.4 Message types

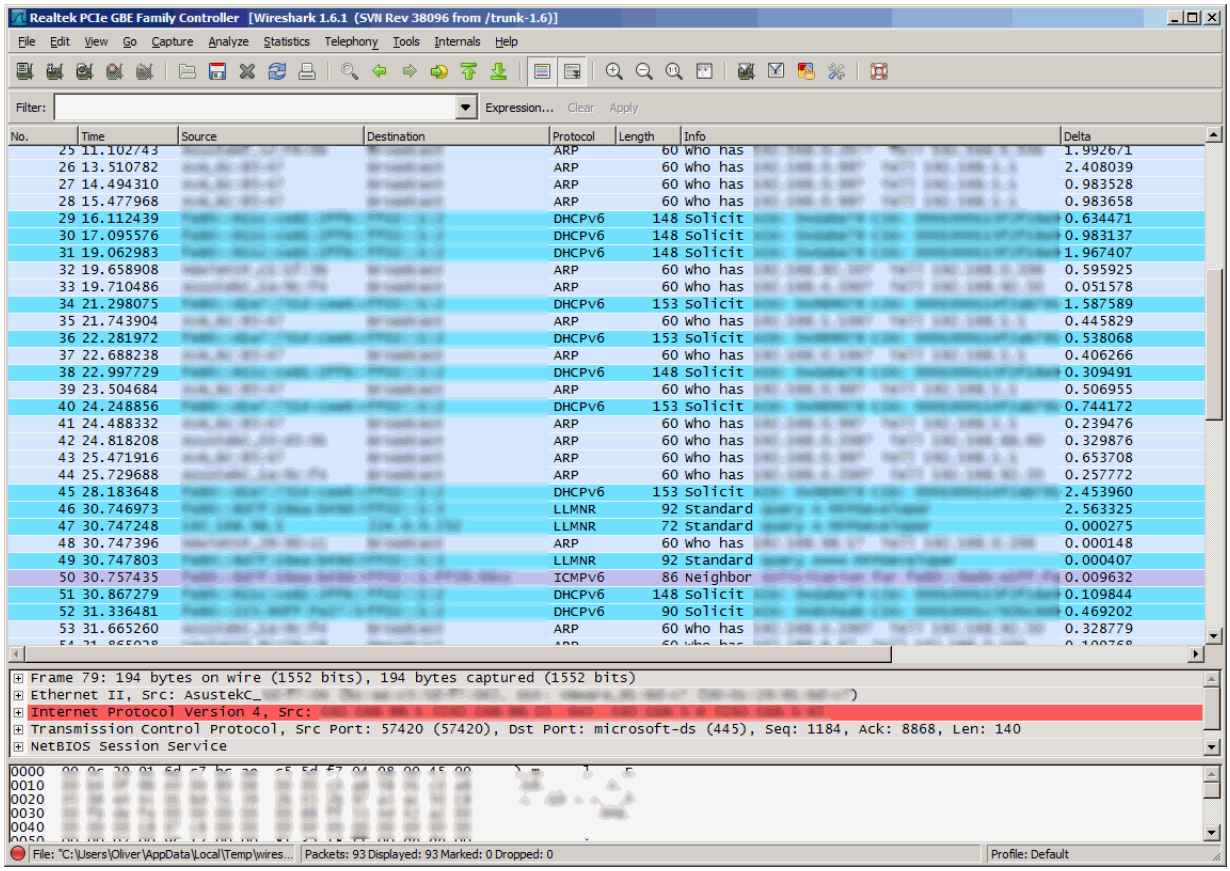
The same message types are used for log and warning messages. Separate filters can be used for both log and warnings. For details, refer to *IP_SetLogFilter* on page 1220 and *IP_SetWarnFilter* on page 1221 as wells as *IP_AddLogFilter* on page 1218 and *IP_AddLogFilter* on page 1218 for more information about using the message types.

Symbolic name	Description
IP_MTYPE_INIT	Activates output of messages from the initialization of the stack that should be logged.
IP_MTYPE_CORE	Activates output of messages from the core of the stack that should be logged.
IP_MTYPE_ALLOC	Activates output of messages from the memory allocating module of the stack that should be logged.
IP_MTYPE_DRIVER	Activates output of messages from the driver that should be logged.
IP_MTYPE_ARP	Activates output of messages from ARP module that should be logged.
IP_MTYPE_IP	Activates output of messages from IP module that should be logged.
IP_MTYPE_TCP_CLOSE	Activates output of messages from TCP module that should be logged when a TCP connection gets closed.
IP_MTYPE_TCP_OPEN	Activates output of messages from TCP module that should be logged when a TCP connection gets opened.
IP_MTYPE_TCP_IN	Activates output of messages from TCP module that should be logged if a TCP packet is received.
IP_MTYPE_TCP_OUT	Activates output of messages from TCP module that should be logged if a TCP packet is sent.
IP_MTYPE_TCP_RTT	Activates output of messages from TCP module regarding TCP roundtrip time.
IP_MTYPE_TCP_RXWIN	Activates output of messages from TCP module regarding peer TCP Rx window size.
IP_MTYPE_TCP	Activates output of messages from TCP that module should be logged.
IP_MTYPE_UDP_IN	Activates output of messages from UDP module that should be logged when a UDP packet is received.
IP_MTYPE_UDP_OUT	Activates output of messages from UDP module that should be logged if a UDP packet is sent.
IP_MTYPE_UDP	Activates output of messages from UDP module that should be logged if a UDP packet is sent or received.
IP_MTYPE_LINK_CHANGE	Activates output of messages regarding to the link change process.
IP_MTYPE_AUTOIP	Activates output of from the AutoIP module that should be logged.
IP_MTYPE_DHCP	Activates output of messages from DHCP client module that should be logged.
IP_MTYPE_DHCP_EXT	Activates output of optional messages from DHCP client module that should be logged.
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.

Symbolic name	Description
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.

38.5 Using a network sniffer to analyze 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 39

OS integration

emNet 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 emNet.

This chapter provides descriptions of the functions required to fully support emNet in multitasking environments.

39.1 OS integration general information

The complexity of the IP/OS Interface depends on the task model selected. Refer to *Tasks and interrupt usage* on page 45 for detailed informations about the different task models. All OS interface functions for embOS are implemented in `IP_OS_embOS.c` which can be found in the sample folder of the shipment.

The IP/OS interface and its functions are not meant to be used in an application. Their purpose is to provide an OS interface for emNet and calling them from an application might confuse the internal mechanics of emNet.

Function	Description
General macros	
`IP_OS_Delay()`	Blocks the calling task for a given time.
`IP_OS_DisableInterrupt()`	Disables interrupts to lock against calls from interrupt routines.
`IP_OS_EnableInterrupt()`	Enables interrupts that have previously been disabled.
`IP_OS_GetTime32()`	Return the current system time in ms; The value will wrap around after approximately 49.7 days; This is taken into account by the stack.
`IP_OS_Init()`	Initialize (create) all objects required for task synchronization.
`IP_OS_Lock()`	The stack requires a single lock, typically a resource semaphore or mutex.
`IP_OS_Unlock()`	Unlocks the single lock, locked by a previous call to `IP_OS_Lock()` and signals the `IP_Task()` if a packet has been freed.
`IP_Task` synchronization	
`IP_OS_SignalNetEvent()`	Wakes the `IP_Task` if it is waiting for a NET-event or timeout in the function `IP_OS_WaitNetEventTimed()`.
`IP_OS_WaitNetEventTimed()`	Called from `IP_Task()` only or alternatively `IP_TASK_WaitForEvent()`.
`IP_RxTask` synchronization	
`IP_OS_SignalRxEvent()`	
`IP_OS_WaitRxEventTimed()`	Called whenever the RxTask handling is idle (no more packets in the In-FIFO).
IP DriverTask (DTask) synchronization	
`IP_OS_SignalDTaskEvent()`	Called by an interrupt from an external module to signal that an events needs to be handled by the DTask.
`IP_OS_WaitDTaskEventTimed()`	Called whenever the DTask handling is idle (no more events to handle).
Application task synchronization	
`IP_OS_WaitItemTimed()`	Suspend a task which needs to wait for a object; This object is identified by a pointer to it and can be of any type, for example a socket.
`IP_OS_SignalItem()`	Sets an object to signaled state, or resumes tasks which are waiting at the event object.

39.1.1 Examples

OS interface routine for embOS

All OS interface routines are implemented in `IP_OS_embOS.c` which is located in the `Sample\IP\OS\` folder of the IP stack.

39.1.2 IP_OS_Delay()

Description

Blocks the calling task for a given time.

Prototype

```
void IP_OS_Delay(unsigned ms);
```

Parameters

Parameter	Description
ms	Time to block in system ticks (typically 1ms).

39.1.3 IP_OS_DisableInterrupt()

Description

Disables interrupts to lock against calls from interrupt routines.

Prototype

```
void IP_OS_DisableInterrupt(void);
```


39.1.4 IP_OS_EnableInterrupt()

Description

Enables interrupts that have previously been disabled.

Prototype

```
void IP_OS_EnableInterrupt(void);
```


39.1.5 IP_OS_GetTime32()

Description

Return the current system time in ms; The value will wrap around after approximately 49.7 days; This is taken into account by the stack.

Prototype

```
U32 IP_OS_GetTime32(void);
```

Return value

U32 timestamp in system ticks (typically 1ms).

39.1.6 IP_OS_Init()

Description

Initialize (create) all objects required for task synchronization. These are 3 events (for IP_Task, IP_RxTask and DriverTask) and one semaphore for protection of critical code which may not be executed from multiple tasks at the same time.

Prototype

```
IP_OS_API *IP_OS_Init(void);
```

Return value

Pointer to the IP_OS API table.

39.1.7 IP_OS_Lock()

Description

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 threads.

Prototype

```
void IP_OS_Lock(void);
```

Additional information

If the entire stack executes from a single task, no functionality is required here.

39.1.8 IP_OS_Unlock()

Description

Unlocks the single lock, locked by a previous call to `IP_OS_Lock()` and signals the `IP_Task()` if a packet has been freed.

Prototype

```
void IP_OS_Unlock(void);
```

Additional information

If the entire stack executes from a single task, no functionality is required here.

39.1.9 IP_OS_SignalNetEvent()

Description

Wakes the `IP_Task` if it is waiting for a NET-event or timeout in the function `IP_OS_WaitNetEventTimed()`.

Prototype

```
void IP_OS_SignalNetEvent(void);
```

Additional information

If the entire stack executes from a single task, no functionality is required here.

39.1.10 IP_OS_WaitNetEventTimed()

Description

Called from `IP_Task()` only or alternatively `IP_TASK_WaitForEvent()` . Blocks until the timeout expires or a NET-event occurs, meaning `IP_OS_SignalNetEvent()` is called from an other task or ISR.

Prototype

```
unsigned IP_OS_WaitNetEventTimed(unsigned Timeout);
```

Parameters

Parameter	Description
`Timeout`	Time [ms] to wait for an event to be signaled. 0 means infinite wait.

Return value

= 0 An event was signaled.
≠ 0 `Timeout`.

Additional information

If the entire stack executes from a single task, no functionality is required here.

39.1.11 IP_OS_SignalRxEvent()

Prototype

```
void IP_OS_SignalRxEvent(void);
```

Additional information

If the entire stack executes from a single task, no functionality is required here.

39.1.12 IP_OS_WaitDTaskEventTimed()

Description

Called whenever the DTask handling is idle (no more events to handle).

Prototype

```
unsigned IP_OS_WaitDTaskEventTimed(unsigned Timeout);
```

Parameters

Parameter	Description
Timeout	Time [ms] to wait for an event to be signaled. 0 means infinite wait.

Return value

= 0 An event was signaled.
≠ 0 Timeout.

Additional information

If the entire stack executes from a single task, no functionality is required here.

39.1.13 IP_OS_SignalDTaskEvent()

Description

Called by an interrupt from an external module to signal that an events needs to be handled by the DTask.

Prototype

```
void IP_OS_SignalDTaskEvent(void);
```

Additional information

If the entire stack executes from a single task, no functionality is required here.

39.1.14 IP_OS_WaitRxEventTimed()

Description

Called whenever the RxTask handling is idle (no more packets in the In-FIFO).

Prototype

```
unsigned IP_OS_WaitRxEventTimed(unsigned Timeout);
```

Parameters

Parameter	Description
Timeout	Time [ms] to wait for an event to be signaled. 0 means infinite wait.

Return value

= 0 An event was signaled.
≠ 0 Timeout.

Additional information

If the entire stack executes from a single task, no functionality is required here.

39.1.15 IP_OS_WaitItemTimed()

Description

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.

Prototype

```
unsigned IP_OS_WaitItemTimed(void     * pWaitItem,
                             unsigned   Timeout);
```

Parameters

Parameter	Description
pWaitItem	Item to wait for.
Timeout	Time [ms] to wait for an event to be signaled. 0 means infinite wait.

Return value

= 0 An event was signaled.
≠ 0 Timeout.

Additional information

Function is called from an application task and is locked in every case.

39.1.16 IP_OS_SignalItem()

Description

Sets an object to signaled state, or resumes tasks which are waiting at the event object.

Prototype

```
void IP_OS_SignalItem(void * pWaitItem);
```

Parameters

Parameter	Description
pWaitItem	Item to signal.

Additional information

Function is called from a task, not an ISR and is locked in every case.

Chapter 40

Performance & resource usage

This chapter covers the performance and resource usage of emNet. It contains information about the memory requirements in typical systems which can be used to obtain sufficient estimates for most target systems.

40.1 emNet Memory footprint

emNet 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.

40.1.1 emNet on 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;

40.1.1.1 ROM usage ARM7

The following table shows the ROM requirement of emNet:

Description	ROM
emNet - complete stack	approximately 19.0 kBytes

40.1.1.2 RAM usage ARM7

The following table shows the RAM requirement of emNet:

Description	RAM
emNet - complete stack w/o buffers	approximately 1.5 kBytes

40.1.2 emNet on 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;

40.1.2.1 ROM usage Cortex-M3

The following table shows the ROM requirement of emNet:

Description	ROM
emNet - complete stack	approximately 19.0 kBytes

The memory requirements of a interface driver is about 1.5 - 2.0Kbytes.

40.1.2.2 RAM usage Cortex-M3

The following table shows the RAM requirement of emNet:

Description	RAM
emNet - complete stack w/o buffers	approximately 1.5 kBytes

40.2 emNet performance

40.2.1 Performance on 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;

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 emNet:

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.

40.2.2 Performance on 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;

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 emNet:

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 41

Appendix A - File system abstraction layer

41.1 File system abstraction layer

This section provides a description of the file system abstraction layer used by emNet applications which require access to a data storage medium. The file system abstraction layer is supplied with the emNet web server and the emNet FTP server.

Three file system abstraction layer implementations are available:

File name	Description
`IP_FS_emFile.c`	Mapping of the emNet file system abstraction layer functions to emFile functions.
`IP_FS_ReadOnly.c`	Implementation of a read-only file system. Typically used in a web server application.
`IP_FS_Linux.c`	Mapping of the emNet file system abstraction layer functions to Linux file I/O functions.
`IP_FS_Win32.c`	Mapping of the emNet file system abstraction layer functions to Windows file I/O functions.

41.2 File system abstraction layer function table

emNet 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);
  int    (*pfForEachDirEntryEx)     (       void * pContext,
                                      const char * sDir,
                                            void (*pf) (void * pContext,
                                                        void * pFileEntry));

} IP_FS_API;
```


Elements of IP_FS_API

Function	Description
Read only file system functions (required)	
pfOpenFile	Pointer to a function that creates/opens a file and returns the handle of these file.
pfCloseFile	Pointer to a function that closes a file.
pfReadAt	Pointer to a function that reads a file.
pfGetLen	Pointer to a function that returns the length of a file.
Directory query operations	
pfForEachDirEntry	Pointer to a function which is called for each directory entry. Obsolete. Use pfForEachDirEntryEx instead.
pfGetDirEntryFileName	Pointer to a function that returns the name of a file entry.
pfGetDirEntryFileSize	Pointer to a function that returns the size of a file.
pfGetDirEntryFileTime	Pointer to a function that returns the time-stamp of a file.
pfGetDirEntryAttributes	Pointer to a function that returns the attributes of a directory entry.
Write file operations	
pfCreate	Pointer to a function that creates a file.
pfDeleteFile	Pointer to a function that deletes a file.
pfRenameFile	Pointer to a function that renames a file.
pfWriteAt	Pointer to a function that writes a file.
Additional directory operations (optional)	
pfMKDir	Pointer to a function that creates a directory.
pfRMDir	Pointer to a function that deletes a directory.
Additional operations (optional)	
pfIsFolder	Pointer to a function that checks if a path is a folder.
pfMove	Pointer to a function that moves a file or directory.
pfForEachDirEntryEx	Pointer to a function which is called for each directory entry.

41.2.1 emFile interface

The emNet 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 emNet 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_Obsolete
  _GetDirEntryFileName,
  _GetDirEntryFileSize,
  _GetDirEntryFileTime,
  _GetDirEntryAttributes,
  //
  // Simple write type file operations.
  //
  _Create,
  _DeleteFile,
  _RenameFile,
  _WriteAt,
  //
  // Additional directory operations
  //
  _MKDir,
  _RMDir,
  //
  // Additional operations
  //
  _IsFolder,
  _Move,
  _ForEachDirEntry
};
```

The emFile interface is used in all SEGGER Eval Packages.

41.2.2 Read-only file system

The emNet 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 emNet 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,
  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 emNet Web server example application.

41.2.3 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 readonly 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_FS_RO
```

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.

41.2.4 Windows file system interface

The emNet 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.
  //
  NULL,
  _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,
  _IP_FS_WIN32_ForEachDirEntry
};
```

The Windows file system interface is supplied with the FTP and the Web server add-on packages. It is used by default with the emNet FTP server application.

Chapter 42

Support

This chapter should help if any problem occurs, e.g. with the hardware or the use of the emNet functions, and describes how to contact the emNet support.

42.1 Contacting support

If you are a registered emNet user and you need to contact the emNet support, please send the following information via email to ticket_emnet@segger.com:

- The emNet version.
- Your emNet registration number.
- If you are unsure about the above information, you may also use the name of the emNet ZIP-file (which contains the above information).
- The configuration files `IP_Config_*.c and IP_Conf.h`
- A detailed description of the problem.
- Optionally, a project with which we can reproduce the problem.

Chapter 43

Glossary

Term	Definition
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.

Term	Definition
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.

