Embedded Systems

CS 397

TRIMESTER 3, AY 2021/22

Ethernet (IEEE 802.3): STM32Cube with LwIP TCP/IP Stack

[Ref_09-3] UM1713, Developing Applications on STM32Cube with LwIP TCP/IP Stack

[Ref_09-3a] Adam Dunkels, Design and Implementation of the LwIP TCP_IP Stack, SICS

Dr. LIAW Hwee Choo

Department of Electrical and Computer Engineering

DigiPen Institute of Technology Singapore

HweeChoo.Liaw@DigiPen.edu

STM32Cube with LwIP

STMCube is a STMicroelectronics initiative to ease developers life by reducing development efforts, time and cost. STM32Cube includes:

- The STM32CubeIDE (STM32CubeMX), a graphical software configuration tool that generates C initialization code using graphical wizards.
- A comprehensive embedded software package, delivered per series (such as STM32CubeF4 for STM32F4 series, STM32CubeF7 for STM32F7 series, …), includes:
  - The STM32Cube HAL, an STM32 abstraction layer embedded software, ensuring maximized portability across STM32 portfolio
  - A consistent set of middleware components such as RTOS, USB, LwIP (TCP/IP), Graphics, . . .
  - All embedded software utilities coming with a full set of examples.

Background for LwIP TCP/IP Stack

- Some STM32 microcontrollers feature a high-quality 10/100 Mbit/s Ethernet peripheral that supports both Media Independent Interface (MII) and **Reduced Media Independent Interface** (**RMII**) to interface with the **Physical Layer** (**PHY**).

- When working with an Ethernet communication interface, a TCP/IP stack is mostly used to communicate over a local or a wide area network.

- The UM1713 is intended for developers who use STM32Cube firmware on STM32 microcontrollers. It provides a full description of how to integrate a free middleware TCP/IP stack using STM32Cube HAL drivers into an embedded application based on STM32 microcontroller.

- The middleware TCP/IP stack is the **LwIP (Lightweight IP)** which is an open source stack intended for embedded devices.

- A dedicated STM32Cube firmware package is provided for each series. It includes Ethernet HAL driver, LwIP middleware and application examples with and without RTOS running on STM32 evaluation boards.

LwIP TCP/IP Stack Features

TCP/IP: Transmission Control Protocol /Internet Protocol
https://www.fortinet.com/resources/cyberglossary/tcp-ip

LwIP is a free TCP/IP stack developed by Adam Dunkels at the Swedish Institute of Computer Science (SICS) and licensed under a modified BSD license (refer UM1713) .

The focus of the LwIP TCP/IP implementation is to reduce RAM usage while keeping a full scale TCP/IP stack. This makes LwIP suitable for use in embedded systems.

LwIP comes with the following protocols:

BSD = Berkeley Software Distribution

- IPv4 and IPv6 (Internet Protocol v4 and v6)
- ICMP (Internet Control Message Protocol) for network maintenance and debugging
- IGMP (Internet Group Management Protocol) for multicast traffic management
- UDP (User Datagram Protocol)
- TCP (Transmission Control Protocol)
- DNS (Domain Name Server)
- SNMP (Simple Network Management Protocol)
- DHCP (Dynamic Host Configuration Protocol)
- PPP (Point to Point Protocol)
- ARP (Address Resolution Protocol)

LwIP V2.1.2 (15 March 2019)

The latest info of LwIP is from files "README" and "st_readme.txt" at

C:\STM32CubeIDE\Repository\STM32Cube_FW_F7_V1.17.0\Middlewares\Third_Party\LwIP

LwIP TCP/IP Stack Features

The source code for the LwIP stack can be downloaded from

http://savannah.nongnu.org

LwIP has three application programming interfaces (APIs):

- **Raw API** is the native LwIP API. It enables the development of applications using event callbacks. This API provides the best performance and optimized code size, but adds some complexity to application development.

- **Netconn API** is a high-level sequential API that requires a real-time operating system (RTOS). The Netconn API enables multithreaded operations.

- **BSD Socket API**: Berkeley-like Socket API (developed on top of the Netconn API)

BSD = Berkeley Software Distribution

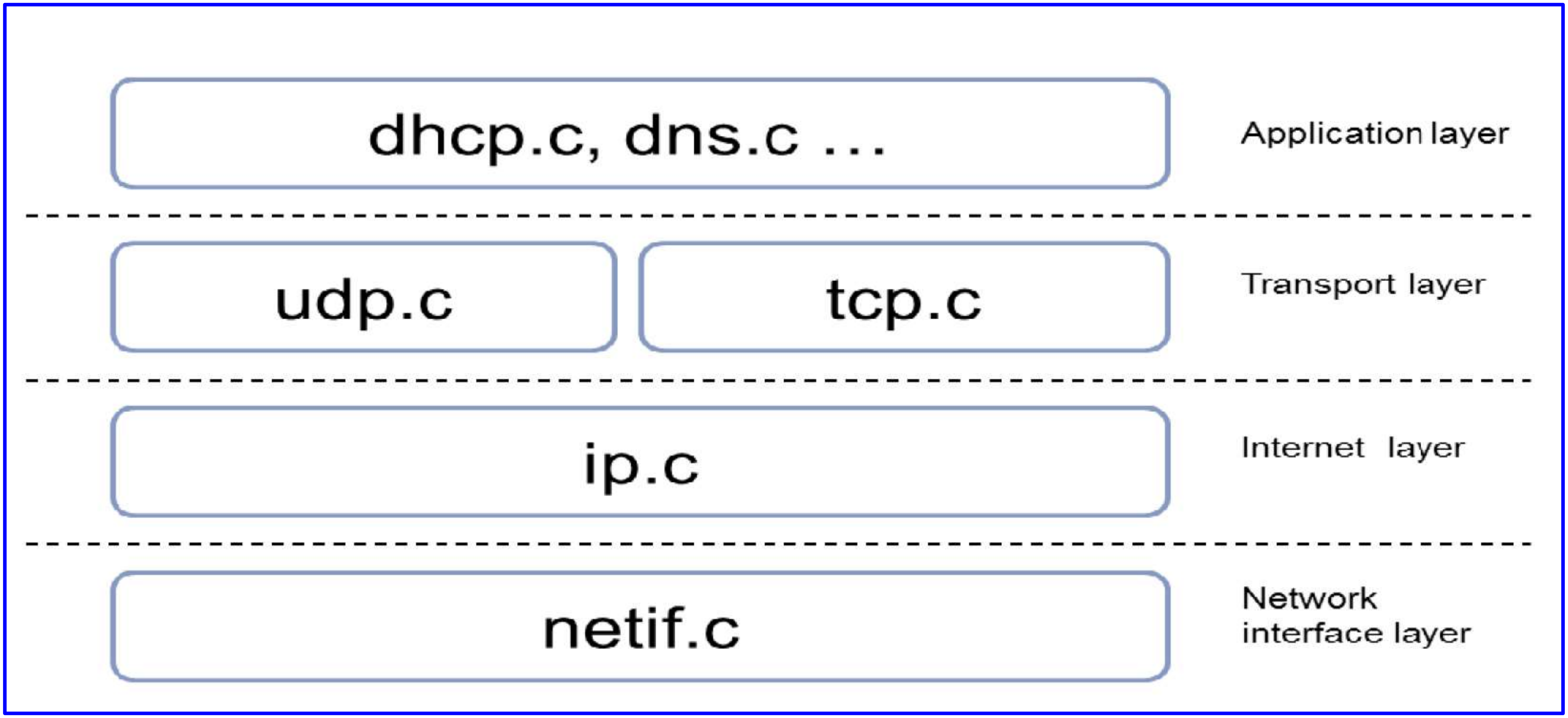
LwIP Architecture

LwIP complies with the TCP/IP model architecture which specifies how data should be formatted, transmitted, routed and received to provide end-to-end communications.

This model includes four abstraction layers which are used to sort all related protocols according to the scope of networking involved (see Figure on next page). From lowest to highest, the layers are:

- **Link layer (network interface layer)** contains communication technologies for a single network segment (link) of a local area network.
- **Internet layer (IP)** connects independent networks, thus establishing internetworking.
- **Transport layer** handles host-to-host communications.
- **Application layer** contains all protocols for specific data communications services on a process-to-process level

LwIP Architecture

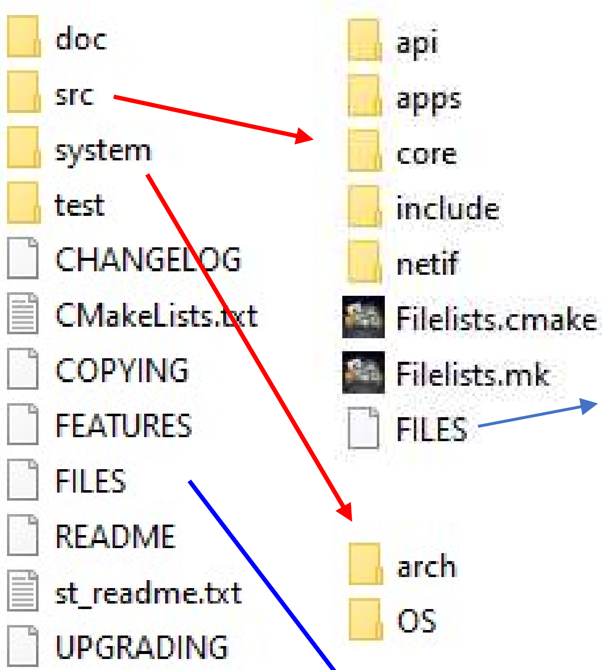


LwIP Architecture

STM32Cube with LwIP

LwIP Stack Folder Organization

C:\STM32CubeIDE\Repository\STM32Cube_FW_F7_V1.17.0\Middlewares\Third_Party\LwIP



api/ - The code for high-level (Netconn & Socket) wrapper API. Not needed if you use the low-level call-back/raw API.

apps/ - Higher layer applications that are specifically programmed with the LwIP low-level raw API.

core/ - The core of the TPC/IP stack; protocol implementations, memory & buffer management, and the low-level raw API.

include/ - LwIP include files

netif/ - Generic network interface device drivers

doc/ - The documentation for LwIP

src/ - The source code for the LwIP TCP/IP stack

system/ - The LwIP port hardware implementation files

- arch/ - contains STM32 architecture port files (used data types,...)
- OS/ - contains LwIP port hardware implementation files using an operating system

test/ - Some code to test whether the sources do what they should

LwIP (Raw, Netconn, Socket) API overview – Raw API

The Raw API is based on the native LwIP API. It is used to develop callback-based applications.

When initializing the application, the user needs to register callback functions to different core events (such as TCP_Sent, TCP_error,...). The callback functions are called from the LwIP core layer when the corresponding event occurs.

API functions		Description
TCP connection setup	tcp_new	Creates a new TCP PCB (protocol control block).
	tcp_bind	Binds a TCP PCB to a local IP address and port.
	tcp_listen	Starts the listening process on the TCP PCB
	tcp_accept	Assigns a callback function that will be called when new TCP connection arrives.
	tcp_accepted	Informs the LwIP stack that an incoming TCP connection has been accepted.
	tcp_connect	Connects to a remote TCP host.
Sending TCP data	tcp_write	Queues up data to be sent.
	tcp_sent	Assigns a callback function that will be called when data are acknowledged by the remote host.
	tcp_output	Forces queued data to be sent.

Summary of TCP Raw API functions

LwIP (Raw, Netconn, Socket) API overview – Raw API

API functions		Description
Receiving TCP	tcp_recv	Sets the callback function that will be called when new data arrives.
	tcp_recved	Must be called when the application has processed the incoming data packet (for TCP window management).
Application polling	tcp_poll	Assigns a callback functions that will be called periodically. It can be used by the application to check if there are remaining application data that needs to be sent or if there are connections that need to be closed.
Closing and aborting connections	tcp_close	Closes a TCP connection with a remote host.
	tcp_err	Assigns a callback function for handling connections aborted by the LwIP due to errors (such as memory shortage errors).
	tcp_abort	Aborts a TCP connection.

Summary of TCP Raw API functions (continued)

LwIP (Raw, Netconn, Socket) API overview – Raw API

API functions	Description
udp_new	Creates a new UDP PCB.
udp_remove	Removes and de-allocates a UDP PCB.
udp_bind	Binds a UDP PCB with a local IP address and port.
udp_connect	Sets up a UDP PCB remote IP address and port.
udp_disconnect	Removes a UDP PCB remote IP and port.
udp_send	Sends UDP data.
udp_recv	Specifies a callback function which is called when a datagram is received.

Summary of UDP Raw API functions (continued)

LwIP (Raw, Netconn, Socket) API overview – Netconn API

The Netconn API is a high-level sequential API, which model of execution is based on the blocking open-read-write-close paradigm.

To operate correctly, this API must run in a multithreaded operating mode implementing a dedicated thread for the LwIP TCP/IP stack and/or multiple threads for the application.

LwIP (Raw, Netconn, Socket) API overview – Netconn API

API functions	Description
netconn_new	Creates a new connection.
netconn_delete	Deletes an existing connection.
netconn_bind	Binds a connection to a local IP address and port.
netconn_connect	Connects to a remote IP address and port.
netconn_send	Sends data to the currently connected remote IP/port (not applicable for TCP connections).
netconn_recv	Receives data from a netconn.
netconn_listen	Sets a TCP connection to a listening mode.
netconn_accept	Accepts an incoming connection on a listening TCP connection.
netconn_write	Sends data on a connected TCP netconn.
netconn_close	Closes a TCP connection without deleting it.

Summary of Netconn API functions

LwIP (Raw, Netconn, Socket) API overview – Socket API

LwIP offers the standard BSD socket API. This is a sequential API which is internally built on top of the Netconn API.

BSD = Berkeley Software Distribution

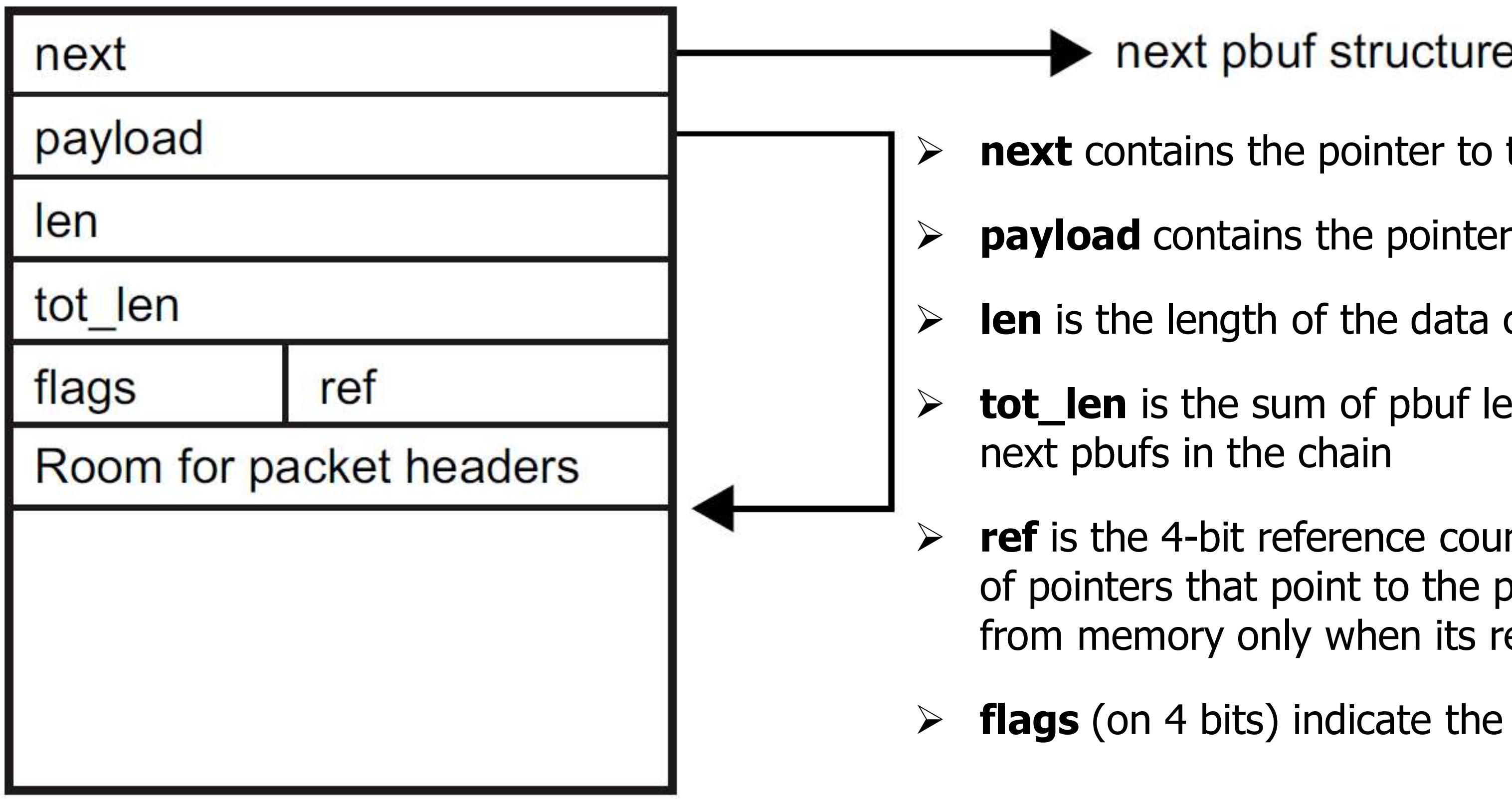
API functions	Description
socket	Creates a new socket.
bind	Binds a socket to an IP address and port.
listen	Listens for socket connections.
connect	Connects a socket to a remote host IP address and port.
accept	Accepts a new connection on a socket.
read	Reads data from a socket.
write	Writes data on a socket.
close	Closes a socket (socket is deleted).

Summary of Socket API functions

LwIP buffer management – Packet Buffer Structure [Ref_09-3a]

- LwIP manages packet buffers using a data structure called **pbuf.**
- The pbuf structure enables the allocation of a dynamic memory to hold a packet content and lets packets reside in the static memory.
- Pbufs can be linked together in a chain, thus enabling packets to span over several pbufs.

The pbuf structure



- **next** contains the pointer to the next pbuf in a pbuf chain
- **payload** contains the pointer to the packet data (payload)
- **len** is the length of the data content of the pbuf
- **tot_len** is the sum of pbuf len plus all the len fields of the next pbufs in the chain
- **ref** is the 4-bit reference count that indicates the number of pointers that point to the pbuf. A pbuf can be released from memory only when its reference count is zero.
- **flags** (on 4 bits) indicate the type of pbuf.

LwIP buffer management – Packet Buffer Structure

LwIP defines three types of pbufs, depending on the allocation type:

- **PBUF_POOL**

  pbuf allocation is performed from a pool of statically pre-allocated pbufs of pre-defined size. Depending on the data size that needs to be allocated, one or multiple chained pbufs are required.

- **PBUF_RAM**

  pbuf is dynamically allocated in memory (one contiguous chunk of memory for the full pbuf)

- **PBUF_ROM**

  No memory space allocation is required for user payload; the pbuf payload pointer points to data in ROM memory that can be used only for sending constant data.

LwIP buffer management – Packet Buffer Structure

For packet reception:

- The suitable pbuf type is **PBUF_POOL**. It allocates memory quickly for the packet received from the pool of pbufs. Depending on the size of the received packet, one or multiple chained pbufs are allocated.
- The PBUF_RAM is not suitable for packet reception because dynamic allocation takes some delay. It may also lead to memory fragmentation.

For packet transmission:

- The user can choose the most suitable pbuf type according to the data to be transmitted.

LwIP buffer management – APIs

LwIP has a specific API for working with pbufs. This API is implemented in the pbuf.c core file.

The pbuf API functions

API functions	Description
pbuf_alloc	Allocates a new pbuf.
pbuf_realloc	Resizes a pbuf (shrink size only).
pbuf_ref	Increments the reference count field of a pbuf.
pbuf_free	Decrements the pbuf reference count. If it reaches zero, the pbuf is deallocated.
pbuf_clen	Returns the count number of pbufs in a pbuf chain.
pbuf_cat	Chains two pbufs together (but does not change the reference count of the tail pbuf chain).
pbuf_chain	Chains two pbufs together (tail chain reference count is incremented).
pbuf_dechain	Unchains the first pbuf from its succeeding pbufs in the chain.
pbuf_header	Adjusts the payload pointer to hide or reveal headers in the payload.
pbuf_copy_partial	Copies (part of) the contents of a packet buffer to an application supplied buffer.
pbuf_take	Copies application supplied data into a pbuf.
pbuf_coalesce	Creates a single pbuf out of a queue of pbufs.
pbuf_memcmp	Compare pbuf contents at specified offset with other memory
pbuf_memfind	Find occurrence of memory in pbuf, starting from an offset
pbuf_strstr	Find occurrence of a string in pbuf, starting from an offset

LwIP buffer management – Remark

- ‘**pbuf**’ can be a single pbuf or a chain of pbufs.
- When working with the Netconn API, **netbufs** (network buffers) are used for sending/receiving data.
- A netbuf is simply a wrapper for a pbuf structure. It can accommodate both allocated and referenced data.
- A dedicated API (implemented in file **netbuf.c**) is provided for managing **netbufs** (allocating, freeing, chaining, extracting data,...).

Interfacing LwIP with STM32Cube Ethernet HAL Driver

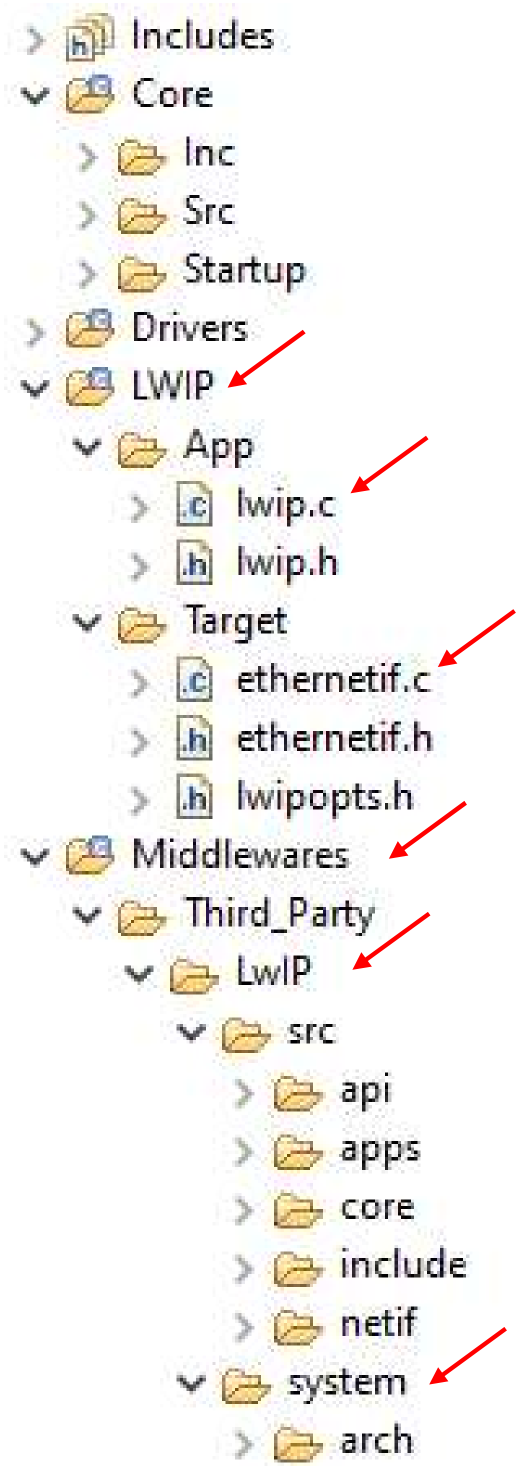
This STMCubeF7 package includes two implementations:

- Implementation without operating system (standalone)
- Implementation with an operating system using **CMSIS-RTOS API**

For both implementations, the **ethernetif.c** file is used to link the **LwIP stack** to the **STM32 Ethernet network interface**.

The port of LwIP stack that must be connected to STM32F7xx is in the "**LwIP/system**" folder.

The Ethernet handle of the HAL (**ETH_HandleTypeDef**) should be declared in the **ethernetif.c** file, as well as the Ethernet DMA descriptors (**ETH_DMADescTypeDef**) and the Rx/Tx buffers of the Ethernet driver.



`Lwip.c: This file provides initialization code for LWIP middleware.`

The LwIP / Ethernet Interface API

Function	Description
low_level_init	Calls the Ethernet driver functions to initialize the STM32F4xx Ethernet peripheral (or STM32F7xx)
low_level_output	Calls the Ethernet driver functions to send an Ethernet packet
low_level_input	Calls the Ethernet driver functions to receive an Ethernet packet.
ethernetif_init	Initializes the network interface structure (netif) and calls low_level_init to initialize the Ethernet peripheral
ethernetif_input	Calls low_level_input to receive a packet then provide it to the LwIP stack

The LwIP / Ethernet interface API functions **(ethernetif.c)**

Initialize the Ethernet peripheral using HAL API in LwIP interface

```
/* ethernetif.c */ // incomplete, highlight of critical code

/* Global Ethernet handle */
ETH_HandleTypeDef heth;

/* LL Driver Interface ( LwIP stack --> ETH)
 * In this function, the hardware should be initialized. Called from ethernetif_init().
 * @param netif the already initialized lwip network interface structure
 * for this ethernetif  */
static void low_level_init(struct netif *netif)
{
    uint32_t regvalue = 0;
    HAL_StatusTypeDef hal_eth_init_status;

    /* Init ETH */
    uint8_t MACAddr[6] ;
    heth.Instance = ETH;
    heth.Init.AutoNegotiation = ETH_AUTONEGOTIATION_ENABLE;
    heth.Init.Speed = ETH_SPEED_100M;
    heth.Init.DuplexMode = ETH_MODE_FULLDUPLEX;
    heth.Init.PhyAddress = LAN8742A_PHY_ADDRESS;
    MACAddr[0] = 0x02;
    MACAddr[1] = 0x80;
    MACAddr[2] = 0xE1;
    MACAddr[3] = 0x00;
    MACAddr[4] = 0x00;
    MACAddr[5] = 0xFF;
```

```c
heth.Init.MACAddr = &MACAddr[0];
heth.Init.RxMode = ETH_RXPOLLING_MODE;
heth.Init.ChecksumMode = ETH_CHECKSUM_BY_HARDWARE;
heth.Init.MediaInterface = ETH_MEDIA_INTERFACE_RMII;

hal_eth_init_status = HAL_ETH_Init(&heth);

if (hal_eth_init_status == HAL_OK)
{
  /* Set netif link flag */
  netif->flags |= NETIF_FLAG_LINK_UP;
}
/* Initialize Tx Descriptors list: Chain Mode */
HAL_ETH_DMATxDescListInit(&heth, DMATxDscrTab, &Tx_Buff[0][0], ETH_TXBUFNB);

/* Initialize Rx Descriptors list: Chain Mode  */
HAL_ETH_DMARxDescListInit(&heth, DMARxDscrTab, &Rx_Buff[0][0], ETH_RXBUFNB);

#if LWIP_ARP || LWIP_ETHERNET
/* set MAC hardware address length */
netif->hwaddr_len = ETH_HWADDR_LEN;
/* set MAC hardware address */
netif->hwaddr[0] =  heth.Init.MACAddr[0];
netif->hwaddr[1] =  heth.Init.MACAddr[1];
netif->hwaddr[2] =  heth.Init.MACAddr[2];
netif->hwaddr[3] =  heth.Init.MACAddr[3];
netif->hwaddr[4] =  heth.Init.MACAddr[4];
netif->hwaddr[5] =  heth.Init.MACAddr[5];
```

stm32f7xx_hal_conf.h

```c
/* 4 Rx buffers of size ETH_RX_BUF_SIZE  */
#define ETH_RXBUFNB  ((uint32_t)4U)
/* 4 Tx buffers of size ETH_TX_BUF_SIZE  */
#define ETH_TXBUFNB  ((uint32_t)4U)
```

```
  /* maximum transfer unit */
  netif->mtu = 1500;

  /* Accept broadcast address and ARP traffic */
  /* don't set NETIF_FLAG_ETHARP if this device is not an ethernet one */
  #if LWIP_ARP
      netif->flags |= NETIF_FLAG_BROADCAST | NETIF_FLAG_ETHARP;
  #else
      netif->flags |= NETIF_FLAG_BROADCAST;
  #endif /* LWIP_ARP */

  /* Enable MAC and DMA transmission and reception */
  HAL_ETH_Start(&heth);

  /* Read Register Configuration */
  HAL_ETH_ReadPHYRegister(&heth, PHY_ISFR, &regvalue);
  regvalue |= (PHY_ISFR_INT4);

  /* Enable Interrupt on change of link status */
  HAL_ETH_WritePHYRegister(&heth, PHY_ISFR , regvalue );

  /* Read Register Configuration */
  HAL_ETH_ReadPHYRegister(&heth, PHY_ISFR , &regvalue);
#endif /* LWIP_ARP || LWIP_ETHERNET */
}
```

Interfacing LwIP with STM32Cube Ethernet HAL Driver – Remark

The **ethernetif_input()** function implementation differs between standalone and RTOS modes:

ethernetif.c

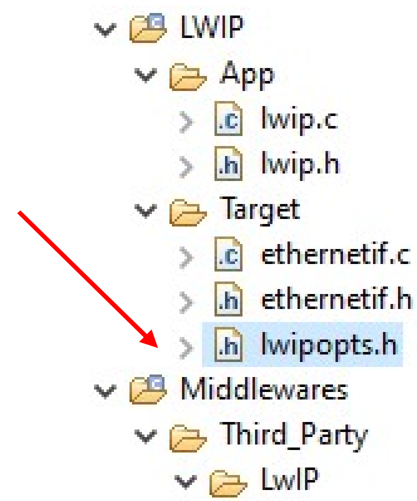
- In standalone applications, this function must be inserted into the main loop of the application to poll for any received packet.
- In RTOS applications, this function is implemented as a thread waiting for a semaphore to handle a received packet. The semaphore is given when the Ethernet peripheral generates an interrupt for a received packet.

The **ethernetif.c** file also implements the Ethernet peripheral MSP routines for low layer initialization (GPIO, CLK, …) and interrupts callbacks.

In case of RTOS implementation, an additional file is used (**sys_arch.c**). This file implements an emulation layer for the RTOS services (message passing through RTOS mailbox, semaphores, etc.). This file should be tailored according to the current RTOS, that is FreeRTOS for this package.

LwIP Configuration

LwIP provides a file named **lwipopts.h** that allows the user to fully configure the stack and all its modules. The user does not need to define all the LwIP options; if an option is not defined, a default value defined in **opt.h** file is used. Therefore, **lwipopts.h** provides a way to override much of the LwIP behavior.



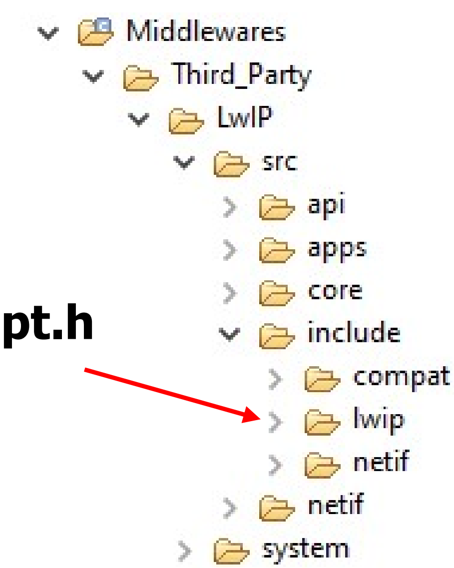
LwIP Modules Support

The user can choose the modules he needs for his application, so that the code size will be optimized by compiling only the selected features.

As an example, to disable UDP and enable DHCP, the following code must be implemented in **lwipopts.h** file:

```
/* Disable UDP */
#define LWIP_UDP 0
/* Enable DHCP */
#define LWIP_DHCP 1
```

opt.h



LwIP Memory Configuration

- LwIP provides a flexible way to manage memory pool sizes and organization.
- It reserves a fixed-size static memory area in the data segment. It is subdivided into the various pools that LwIP uses for the various data structures.
- There is a pool for struct **tcp_pcb**, and another pool for struct **udp_pcb**.
- Each pool can be configured to hold a fixed number of data structures.
- For example, **MEMP_NUM_TCP_PCB** and **MEMP_NUM_UDP_PCB** define the maximum number of **tcp_pcb** and **udb_pcb** structures that can be active in the system at a given time.
- All RAM options are defined in **opt.h** and user options can be added in **lwipopts.h**.
- A summary of the main RAM options is provided as follows:

PCB: Protocol Control Block

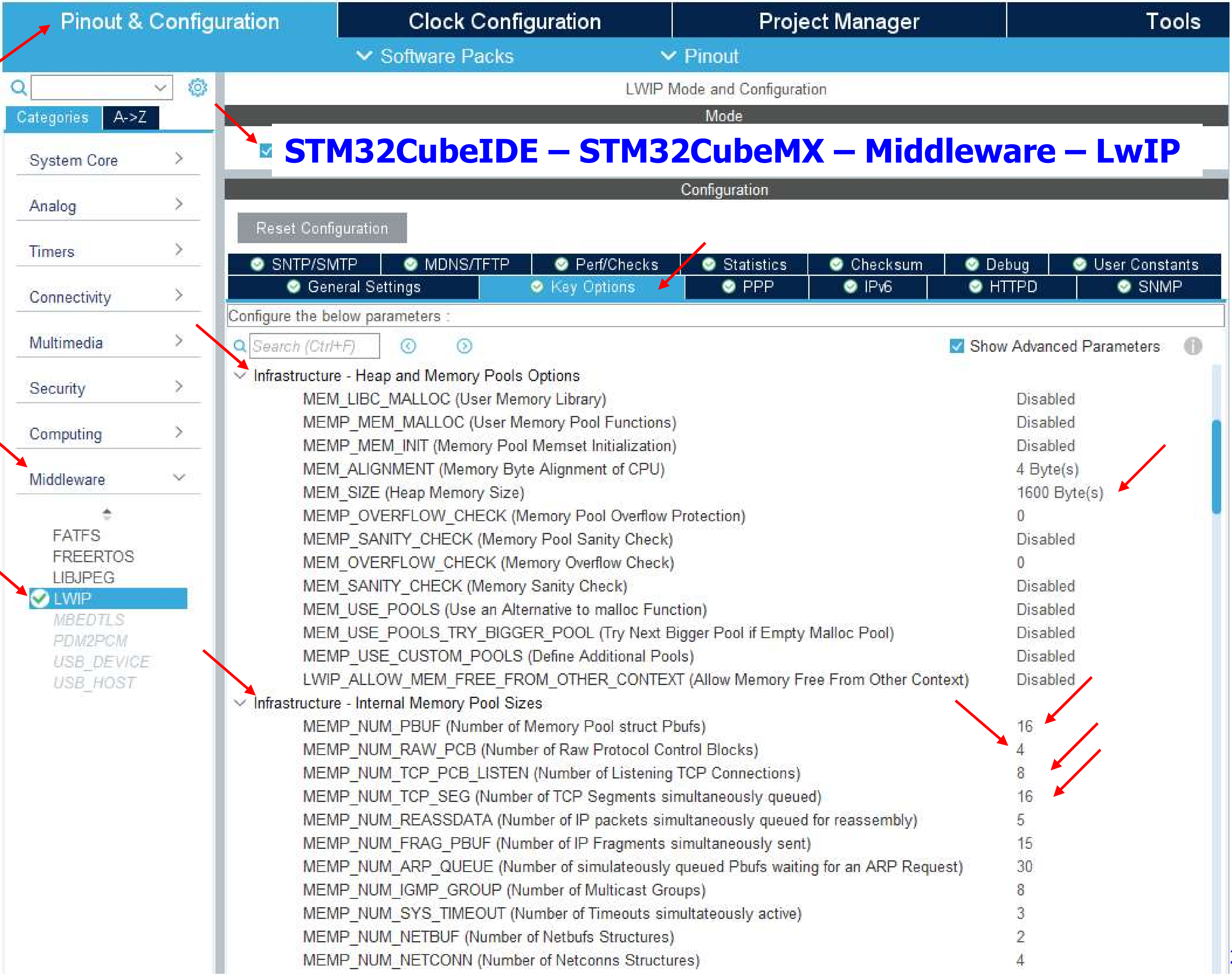
MEMP: MEMory Pool

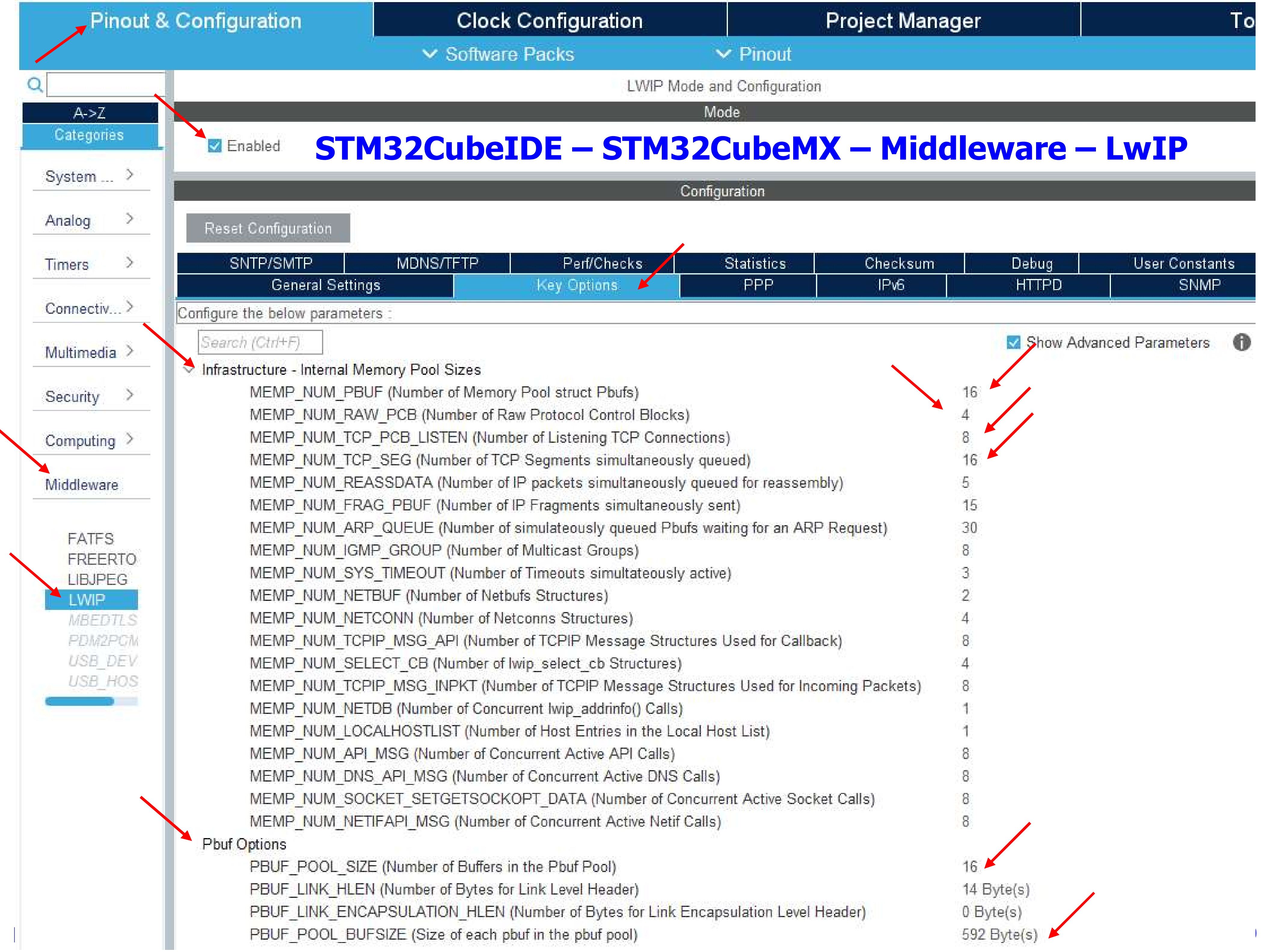
LwIP Memory Configuration – User options can be added in **lwipopts.h**

LwIP memory option	Definition
MEM_SIZE	LwIP heap memory size: used for all LwIP dynamic memory allocations.
MEMP_NUM_PBUF	Total number of MEM_REF and MEM_ROM pbufs.
MEMP_NUM_UDP_PCB	Total number of UDP PCB structures.
MEMP_NUM_TCP_PCB	Total number of TCP PCB structures.
MEMP_NUM_TCP_PCB_LISTEN	Total number of listening TCP PCBs.
MEMP_NUM_TCP_SEG	Maximum number of simultaneously queued TCP segments.
PBUF_POOL_SIZE	Total number of PBUF_POOL type pbufs.
PBUF_POOL_BUFSIZE	Size of a PBUF_POOL type pbufs.
TCP_MSS	TCP maximum segment size.
TCP_SND_BUF	TCP send buffer space for a connection.
TCP_SND_QUEUELEN	Maximum number of pbufs in the TCP send queue.
TCP_WND	Advertised TCP receive window size.

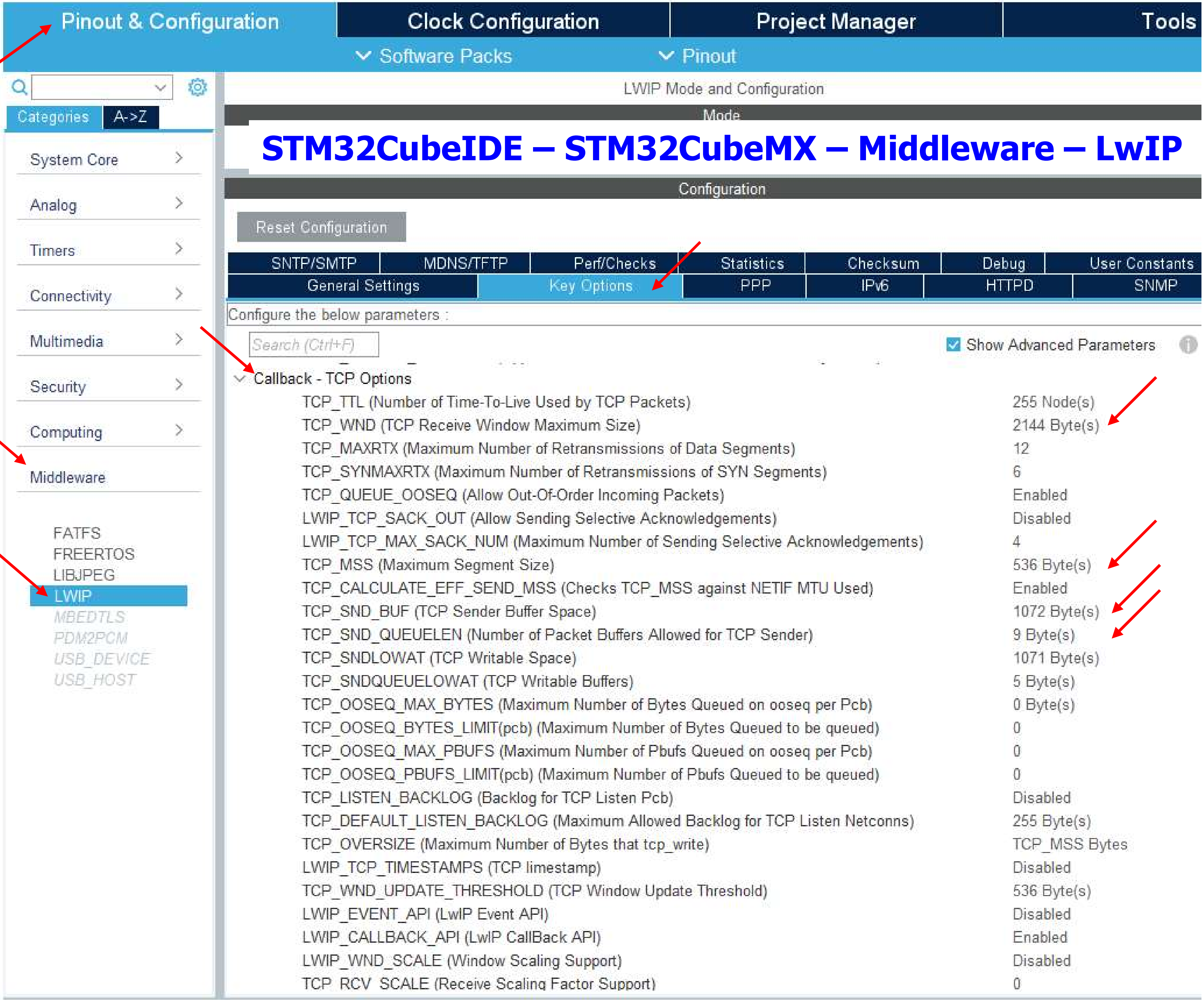
RAW

STM32CubeIDE – STM32CubeMX – Middleware – LwIP




Pinout & Configuration
Clock Configuration
Project Manager
To
Software Packs
Pinout
LWIP Mode and Configuration
Mode
A->Z
Categories
Enabled
STM32CubeIDE – STM32CubeMX – Middleware – LwIP
System ...
Analog
Timers
Connectiv...
Multimedia
Security
Computing
Middleware
FATFS
FREERTO
LIBJPEG
LWIP
MBEDTLS
PDM2PCM
USB_DEV
USB_HOS
Configuration
Reset Configuration
SNTP/SMTP
MDNS/TFTP
Perf/Checks
Statistics
Checksum
Debug
User Constants
General Settings
Key Options
PPP
IPv6
HTTPD
SNMP
Configure the below parameters :
Search (Ctrl+F)
Show Advanced Parameters
Infrastructure - Internal Memory Pool Sizes
MEMP_NUM_PBUF (Number of Memory Pool struct Pbufs) 16
MEMP_NUM_RAW_PCB (Number of Raw Protocol Control Blocks) 4
MEMP_NUM_TCP_PCB_LISTEN (Number of Listening TCP Connections) 8
MEMP_NUM_TCP_SEG (Number of TCP Segments simultaneously queued) 16
MEMP_NUM_REASSDATA (Number of IP packets simultaneously queued for reassembly) 5
MEMP_NUM_FRAG_PBUF (Number of IP Fragments simultaneously sent) 15
MEMP_NUM_ARP_QUEUE (Number of simulateously queued Pbufs waiting for an ARP Request) 30
MEMP_NUM_IGMP_GROUP (Number of Multicast Groups) 8
MEMP_NUM_SYS_TIMEOUT (Number of Timeouts simultateously active) 3
MEMP_NUM_NETBUF (Number of Netbufs Structures) 2
MEMP_NUM_NETCONN (Number of Netconns Structures) 4
MEMP_NUM_TCPIP_MSG_API (Number of TCPIP Message Structures Used for Callback) 8
MEMP_NUM_SELECT_CB (Number of lwip_select_cb Structures) 4
MEMP_NUM_TCPIP_MSG_INPKT (Number of TCPIP Message Structures Used for Incoming Packets) 8
MEMP_NUM_NETDB (Number of Concurrent lwip_addrinfo() Calls) 1
MEMP_NUM_LOCALHOSTLIST (Number of Host Entries in the Local Host List) 1
MEMP_NUM_API_MSG (Number of Concurrent Active API Calls) 8
MEMP_NUM_DNS_API_MSG (Number of Concurrent Active DNS Calls) 8
MEMP_NUM_SOCKET_SETGETSOCKOPT_DATA (Number of Concurrent Active Socket Calls) 8
MEMP_NUM_NETIFAPI_MSG (Number of Concurrent Active Netif Calls) 8
Pbuf Options
PBUF_POOL_SIZE (Number of Buffers in the Pbuf Pool) 16
PBUF_LINK_HLEN (Number of Bytes for Link Level Header) 14 Byte(s)
PBUF_LINK_ENCAPSULATION_HLEN (Number of Bytes for Link Encapsulation Level Header) 0 Byte(s)
PBUF_POOL_BUFSIZE (Size of each pbuf in the pbuf pool) 592 Byte(s)

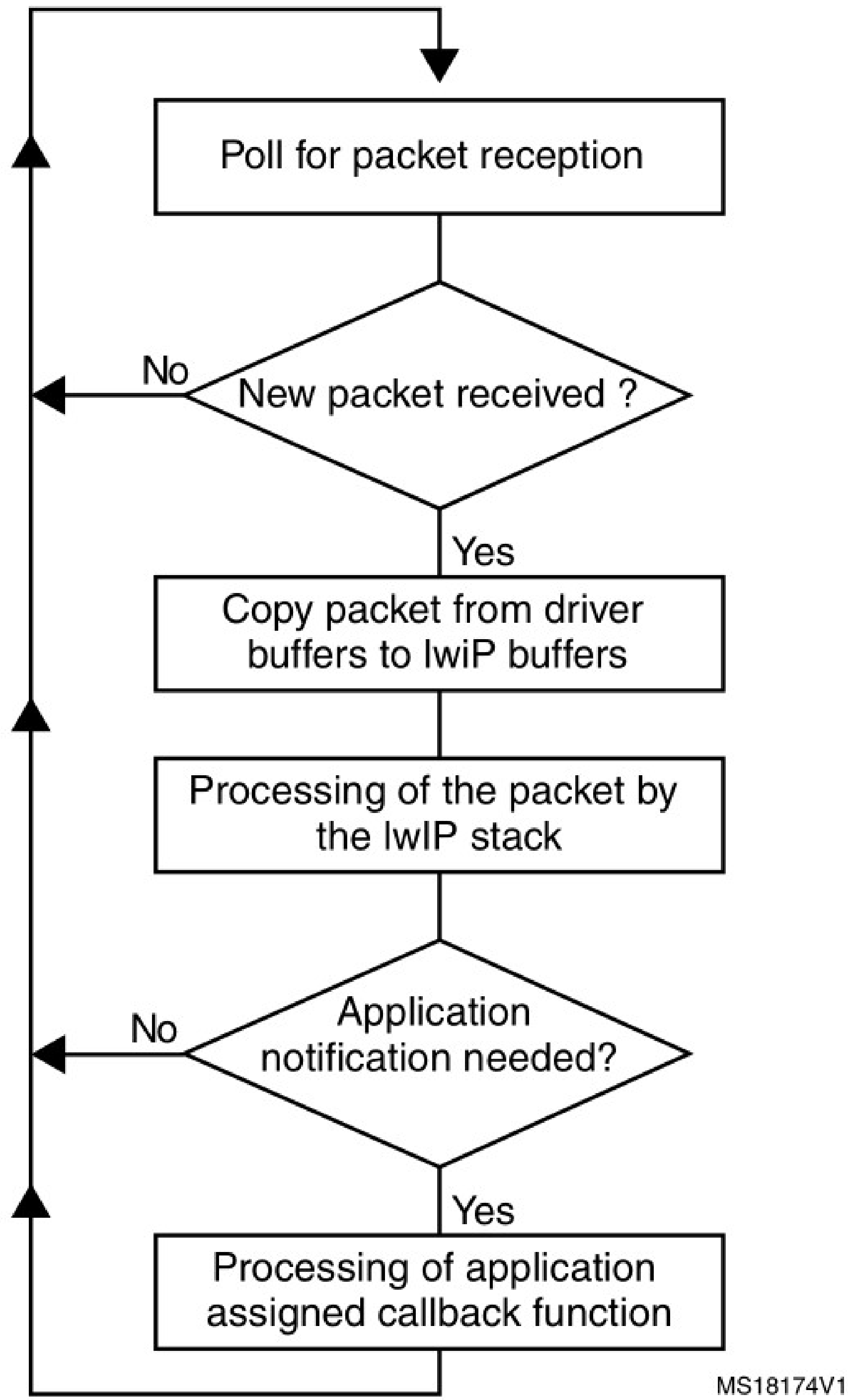
STM32CubeIDE – STM32CubeMX – Middleware – LwIP



Develop Applications in Standalone Mode using Raw API – Operation Model

- In standalone mode (without RTOS), the operation model is based on continuous software polling to check if a packet has been received.
- When a packet has been received, it is first copied from the Ethernet driver buffers into the LwIP buffers.
- To copy the packet as fast as possible, the LwIP buffers (pbufs) should be allocated from the pool of buffers (PBUF_POOL).
- When a packet has been copied, it is handed to the LwIP stack for processing.
- Depending on the received packet, the stack may or may not notify the application layer.
- LwIP communicates with the application layer using event callback functions.
- These functions should be assigned before starting of the communication process.
- Refer next page for the standalone operation model flowchart.

The Standalone Operation Model Flowchart



Develop Applications in Standalone Mode using Raw API – TCP Applications

For TCP applications, the following common callback functions must be assigned:

- Callback for incoming **TCP connection** event, assigned by **TCP_accept** API call
- Callback for incoming **TCP data packet** event, assigned by **TCP_recev** API call
- Callback for signaling **successful data transmission**, assigned by **TCP_sent** API call
- Callback for signaling **TCP error** (after a TCP abort event), assigned by **TCP_err** API call
- **Periodic callback** (every 1 or 2 s) for polling the application, assigned by **TCP_poll** API call

Example of TCP Echo Server Demonstration

The TCP echo server example, provided in the "Hands-On_4-2_LwIP_TCP_Echo_Server", is a simple application that implements a TCP server, which echoes any received TCP data packet coming from a remote client.

The following example provides a description of the firmware structure. This is an extract of the main.c file.

```
int main(void)
{
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART3_UART_Init();
  MX_LWIP_Init();
  /* USER CODE BEGIN 2 */
  tcp_echoserver_init();
  /* USER CODE END 2 */
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    MX_LWIP_Process();
    /* USER CODE END WHILE */
  }
}
```

called to initialize the TCP echo server application

```
/* Initialize the LwIP stack internal
structures, start stack operations, and
configure the network interface (netif) */
```

```
/* Read a received packet from the
Ethernet buffers and send it to the
lwIP for handling */
ethernetif_input(&gnetif);
/* Handle LwIP timeouts */
sys_check_timeouts();
```

```
/* lwip.c */

/* LwIP initialization function */
void MX_LWIP_Init(void)
{
  /* IP addresses initialization */
  IP_ADDRESS[0] = 192;
  IP_ADDRESS[1] = 168;
  IP_ADDRESS[2] = 1;
  IP_ADDRESS[3] = 205;
  NETMASK_ADDRESS[0] = 255;
  NETMASK_ADDRESS[1] = 255;
  NETMASK_ADDRESS[2] = 255;
  NETMASK_ADDRESS[3] = 0;
  GATEWAY_ADDRESS[0] = 192;
  GATEWAY_ADDRESS[1] = 168;
  GATEWAY_ADDRESS[2] = 1;
  GATEWAY_ADDRESS[3] = 1;

  /* Initilialize the LwIP stack without RTOS */
  lwip_init();

  /* IP addresses initialization without DHCP (IPv4) */
  IP4_ADDR(&ipaddr, IP_ADDRESS[0], IP_ADDRESS[1], IP_ADDRESS[2], IP_ADDRESS[3]);
  IP4_ADDR(&netmask, NETMASK_ADDRESS[0], NETMASK_ADDRESS[1] , NETMASK_ADDRESS[2], NETMASK_ADDRESS[3]);
  IP4_ADDR(&gw, GATEWAY_ADDRESS[0], GATEWAY_ADDRESS[1], GATEWAY_ADDRESS[2], GATEWAY_ADDRESS[3]);
```

STM32Cube with LwIP

```c
  /* add the network interface (IPv4/IPv6) without RTOS */
  netif_add(&gnetif, &ipaddr, &netmask, &gw, NULL, &ethernetif_init, &ethernet_input);

  /* Registers the default network interface */
  netif_set_default(&gnetif);

  if (netif_is_link_up(&gnetif))
  {
    /* When the netif is fully configured this function must be called */
    netif_set_up(&gnetif);
  }
  else
  {
    /* When the netif link is down this function must be called */
    netif_set_down(&gnetif);
  }

  /* Set the link callback function, this function is called on change of link status*/
  netif_set_link_callback(&gnetif, ethernetif_update_config);

}
```

Example of TCP Echo Server Demonstration

Refer to source code developed in Hands-On_4-2_LwIP_TCP_Echo_Server

The **tcp_echoserver_init function** (in **tcp_echoserver.c**) description:

1. LwIP API calls **tcp_new** to allocate a new TCP protocol control block (PCB) (**tcp_echoserver_pcb**).
2. The allocated TCP PCB is bound to a local IP address and port using **tcp_bind** function.
3. After binding the TCP PCB, **tcp_listen** function is called in order to start the TCP listening process on the TCP PCB.
4. Finally a **tcp_echoserver_accept** callback function should be assigned to handle incoming TCP connections on the TCP PCB. This is done by using **tcp_accept** LwIP API function.
5. Starting from this point, the TCP server is ready to accept any incoming connection from remote clients.

```c
/* tcp_echoserver.c */
/* @brief  Initializes the tcp echo server  */
void tcp_echoserver_init(void)
{
  /* create new tcp pcb */
  tcp_echoserver_pcb = tcp_new();

  if (tcp_echoserver_pcb != NULL)
  {
    err_t err;

    /* bind echo_pcb to port 7 (ECHO protocol) */
    err = tcp_bind(tcp_echoserver_pcb, IP_ADDR_ANY, SERVER_PORT);

    if (err == ERR_OK)
    {
      /* start tcp listening for echo_pcb */
      tcp_echoserver_pcb = tcp_listen(tcp_echoserver_pcb);

      /* initialize LwIP tcp_accept callback function */
      tcp_accept(tcp_echoserver_pcb, tcp_echoserver_accept);
    }
    else
    {
      /* deallocate the pcb */
      memp_free(MEMP_TCP_PCB, tcp_echoserver_pcb);
    }
  }
}
```

Example of TCP Echo Server Demonstration

The **tcp_echoserver_accept** (in **tcp_echoserver.c**) user callback function description:

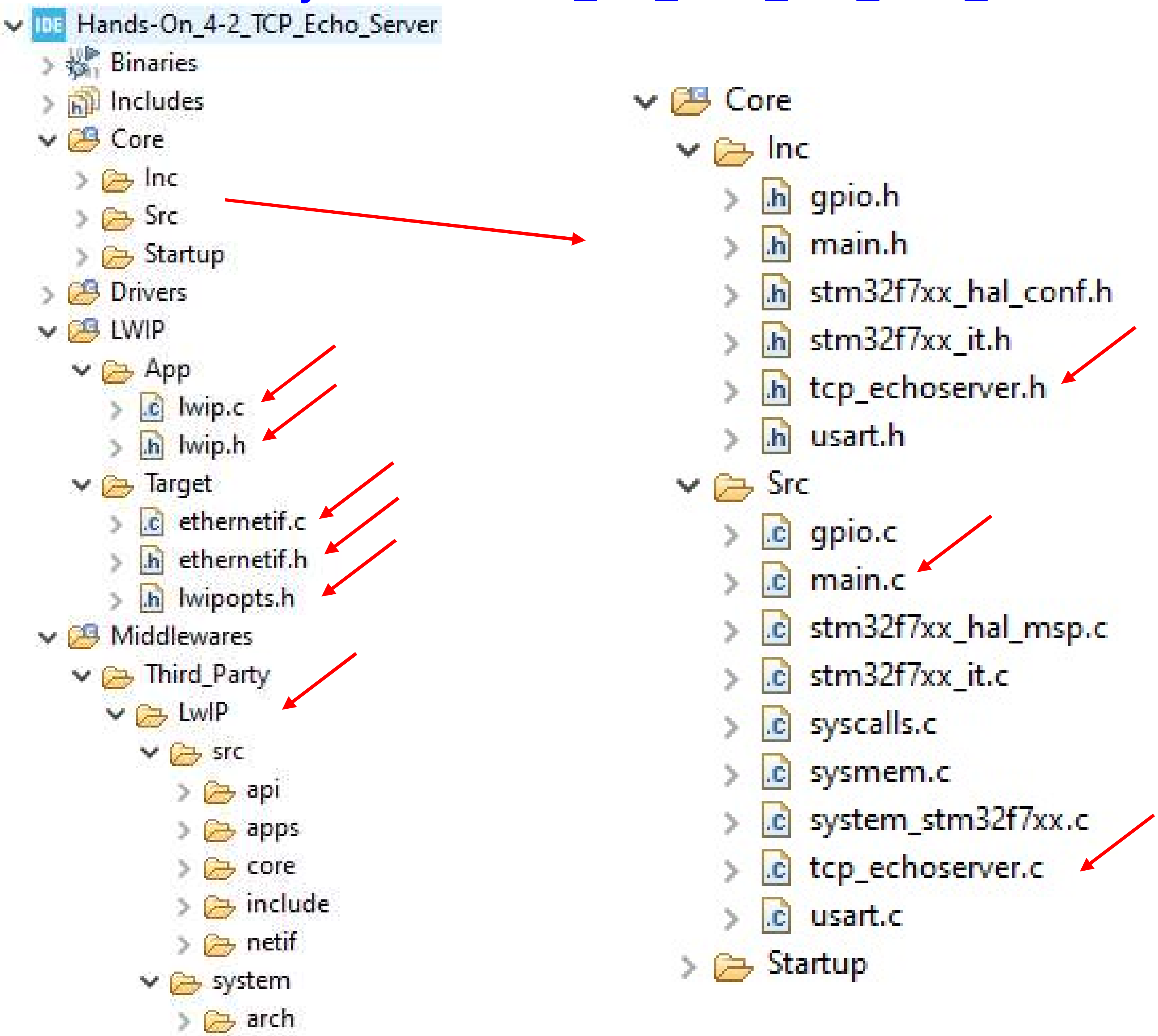
The following functions are called:

1. The new TCP connection is passed to **tcp_echoserver_accept** callback function through **newpcb** parameter.
2. An **es** structure is used to store the application status. It is passed as an argument to the TCP PCB "**newpcb**" connection by calling **tcp_arg** LwIP API.
3. A TCP receive callback function, **tcp_echoserver_recv**, is assigned by calling LwIP API **tcp_recv**. This callback handles all the data traffic with the remote client.
4. A TCP error callback function, **tcp_echoserver_error**, is assigned by calling LwIP API **tcp_err**. This callback handles TCP errors.
5. A TCP poll callback function, **tcp_echoserver_poll**, is assigned by calling LwIP API **tcp_poll** to handle periodic application tasks (such as checking if the application data remains to be transmitted).

Refer to source code developed in Hands-On_4-2_LwIP_TCP_Echo_Server

```c
/*  @brief  This function is the implementation of tcp_accept LwIP callback
  * @param  arg, err : not used
  * @param  newpcb: pointer on tcp_pcb struct for the newly created tcp connection
  * @retval err_t: error status */
static err_t tcp_echoserver_accept(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    err_t ret_err;
    struct tcp_echoserver_struct *es;
    /* set priority for the newly accepted tcp connection newpcb */
    tcp_setprio(newpcb, TCP_PRIO_MIN);
    /* allocate structure es to maintain tcp connection informations */
    es = (struct tcp_echoserver_struct *)mem_malloc(sizeof(struct tcp_echoserver_struct));
    if (es != NULL)
    {
        es->state = ES_ACCEPTED;
        es->pcb = newpcb;
        es->retries = 0;
        es->p = NULL;
        /* pass newly allocated es structure as argument to newpcb */
        tcp_arg(newpcb, es);
        /* initialize lwip tcp_recv callback function for newpcb  */
        tcp_recv(newpcb, tcp_echoserver_recv);
        /* initialize lwip tcp_err callback function for newpcb  */
        tcp_err(newpcb, tcp_echoserver_error);
        /* initialize lwip tcp_poll callback function for newpcb */
        tcp_poll(newpcb, tcp_echoserver_poll, 0);
        ret_err = ERR_OK;
    }
    else
    {
        /*  close tcp connection */
        tcp_echoserver_connection_close(newpcb, es);
        /* return memory error */
        ret_err = ERR_MEM;
    }
    return ret_err;
}
```

Folders and Files of Project “Hands-On_4-2_LwIP_TCP_Echo_Server”



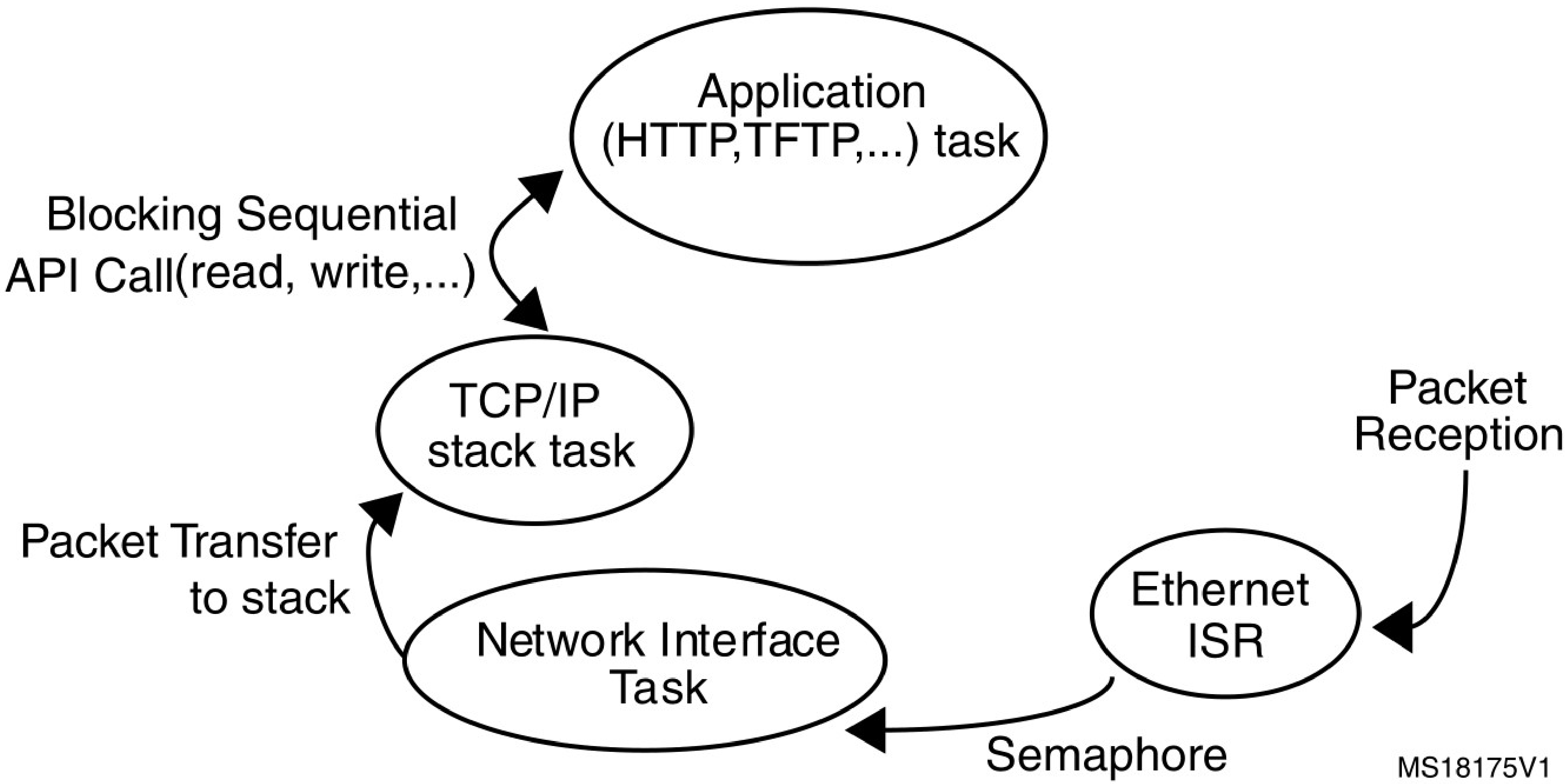
Develop Applications with RTOS using Netconn or Socket API

The **operation model** when working with an RTOS has the following characteristics:

- The TCP/IP stack and the application run in separate threads.
- The application communicates with the stack through sequential API calls that use the RTOS mailbox mechanism for inter-process communications.
- The API calls are blocking calls. This means that the application thread is blocked until a response is received from the stack.
- An additional thread, the network interface thread, is used to get any received packets from driver buffers and provide them to the TCP/IP stack using the RTOS mailbox.
- This thread is informed of a packet reception using the Ethernet receive interrupt service routine.
- Refer to figure for a description of the LwIP operation model flowchart with RTOS.

Develop Applications with RTOS using Netconn or Socket API

The LwIP operation model with RTOS



LwIP Related Package Description and Directories

The STM32CubeF7 software package contains a set of applications (examples) running on top of the LwIP stack, STM32Cube HAL, and BSP drivers. The firmware is composed from the following modules:

- **Drivers**: contains the low level drivers of STM32F7xx microcontroller
  - CMSIS
  - BSP drivers
  - HAL drivers
- **Middlewares** contain libraries and protocol components
  - LwIP TCP/IP stack
  - FatFS
  - FreeRTOS

STM32Cube with LwIP

- **Projects (examples)** contain the source files & configurations of the following applications:
  - Applications running in standalone mode (without an RTOS) based on Raw API:
    - A Web server
    - A TFTP server
    - A TCP echo client application
    - A TCP echo server application
    - A UDP echo client application
    - A UDP echo server application
  - Applications running with the FreeRTOS operating system:
    - A Web server based on netconn API
    - A Web server based on socket API
    - A TCP/UDP echo server application based on netconn API.

Applications are located under Projects repository following this path:

C:\STM32CubeIDE\Repository\STM32Cube_FW_F7_V1.17.0\Projects\

- STM32756G_EVAL\Applications\LwIP\ …

- STM32F769I_EVAL\Applications\LwIP\ … - STM32F767ZI-Nucleo\Applications\LwIP\ …

LwIP Related Package Description and Directories

C:\STM32CubeIDE\Repository\STM32Cube_FW_F7_V1.17.0\Projects\STM32756G_EVAL\Applications\LwIP

- LwIP_HTTP_Server_Netconn_RTOS
- LwIP_HTTP_Server_Raw
- LwIP_HTTP_Server_Socket_RTOS
- LwIP_IAP
- LwIP_TCP_Echo_Client
- LwIP_TCP_Echo_Server
- LwIP_TFTP_Server
- LwIP_UDP_Echo_Client
- LwIP_UDP_Echo_Server
- LwIP_UDPTCP_Echo_Server_Netconn_RTOS

C:\STM32CubeIDE\Repository\STM32Cube_FW_F7_V1.17.0\Projects\STM32F769I_EVAL\Applications\LwIP

- LwIP_HTTP_Server_Netconn_RTOS
- LwIP_HTTP_Server_Raw
- LwIP_HTTP_Server_Socket_RTOS
- LwIP_IAP
- LwIP_StreamingServer

C:\STM32CubeIDE\Repository\STM32Cube_FW_F7_V1.17.0\Projects\STM32F767ZI-Nucleo\Applications\LwIP

- LwIP_HTTP_Server_Netconn_RTOS

Using the LwIP Applications

The STM32Cube LwIP package comes with several applications that use the different LwIP stack API sets. The applications are divided into three categories as shown below:

Categories	Applications
Getting started (basic)	TCP Echo client
	TCP Echo server
	UDP Echo client
	UDP Echo server
	TCP and UDP Echo server (Netconn API)
Features	HTTP Server (Raw API)
	HTTP Server (Netconn API)
	HTTP Server (Socket API)
Integrated	TFTP Server

- **Getting started** applications use the minimal configuration to run applications on top of the LwIP stack.
- **Features** applications provide more flexibility and options supporting network protocols like HTTP & DHCP.
- **Integrated** application supports FatFS middleware component and TFTP protocol to transfer files to and from microSD card located on the evaluation board.