AN3000
Application note

Configuring the NicheLite™ TCP/IP stack for the STM32F107xx microcontroller

Introduction

STM32F107xx connectivity line microcontrollers feature a complete 10/100 Ethernet MAC supporting MII and RMII with hardware support for the IEEE1588 precise time protocol, enabling Ethernet connectivity for real-time applications.

The NicheLite TCP/IP stack, from InterNiche Technologies, supports STM32F107xx connectivity line microcontrollers and is provided free-of-charge to ST customers. This stack is a scaled down version of the full stack available from InterNiche. For download information, please refer to STMicroelectronics web site www.st.com

The objective of this application note is to introduce the NicheLite stack for STM32F107xx and describe the integration considerations to be taken into account when developing a user application including the performance and memory size aspects.

This application note is structured as follows:

- A short glossary is provided in *Section 1*
- A general introduction to TCP/IP is presented in *Section 2*
- The *Section 3* introduces the NicheLite stack
- *Section 4* and *5* describe the NicheLite stack configuration and optimization
- NicheLite stack debugging tips are given in *Section 6*
- *Section 7* presents the NicheTask OS and how to use it in your application
- Lastly, *Section 8* describes the NicheLite package for STM32F107xx and the available demonstrations

This application note is intended to complement the documentation provided by InterNiche, which should be referred to for detailed information.

Contents

List of tables

List of figures

1 Glossary

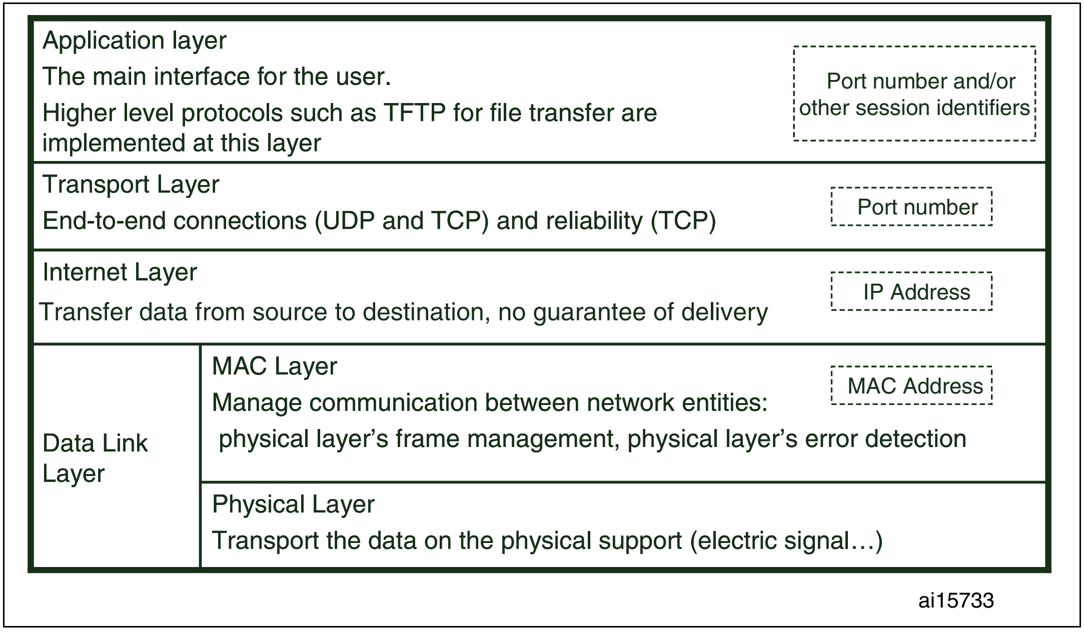
- **ARP** Address resolution protocol
- **BSD** Berkeley software distribution
- **BOOTP** Bootstrap protocol
- **DHCP** Dynamic host configuration protocol
- **DNS** Domain name System
- **FTP** File transfer protocol
- **HTTP** Hypertext transfer protocol
- **ICMP** Internet control message protocol
- **IP** Internet protocol
- **POP3** Post office protocol version 3
- **PPP** Point to point protocol
- **SMTP** Simple mail transfer protocol
- **SNMP** Simple network management protocol
- **Socket** A socket is a combination of an IP address and a port number which is mapped to the application protocol.
- **TCP** Transmission control protocol
- **Telnet** Remote terminal protocol
- **TFTP** Trivial file transfer protocol
- **UDP** User datagram protocol

2 TCP/IP introduction

2.1 TCP/IP a layered protocol stack

The TCP/IP stack is organized in several layers. *Figure 1* below presents this layered architecture.

Figure 1. Protocol stack layers



Application layer
The main interface for the user.
Higher level protocols such as TFTP for file transfer are implemented at this layer
Port number and/or other session identifiers

Transport Layer
End-to-end connections (UDP and TCP) and reliability (TCP)
Port number

Internet Layer
Transfer data from source to destination, no guarantee of delivery
IP Address

Data Link Layer

MAC Layer
Manage communication between network entities:
physical layer's frame management, physical layer's error detection
MAC Address

Physical Layer
Transport the data on the physical support (electric signal…)

ai15733

2.1.1 The application layer

All the upper layer protocols such as HTTP and FTP are located in this layer, these protocols offer higher level features such as file transfer, configuration, network management to the user application. Depending of their characteristics, these protocols rely on UDP, TCP or other protocols

2.1.2 The transport layer

The transport layer provides an end to end connection service. End to end communication means that one (or more) applications on one device can send data to one (or more) applications on other device(s).

One or more port numbers are assigned to each application allowing the transport layer to differentiate the applications running on the same device. The device is identified using the IP address (see internet layer description), the combination of IP address + port number (also called a socket) allows data to be addressed to a specific application running on a specific device.

Unlike the UDP protocol that does not guarantee the delivery; the TCP protocol ensures the reliability of the connection with re-sequencing and flow control mechanisms.

Note: A socket is a combination of an IP address and a port number which is mapped to the application protocol.

2.1.3 The Internet layer

This layer allows transferring data from a source to a destination on separate networks, to perform this inter-network communication this layer uses the IP address (named after the IP protocol). The IP protocol provides a way to send data from a source to a destination but does not guarantee the delivery of the data, it is called a connectionless service.

2.1.4 Link layer

The Link layer can be subdivided in 2 parts, the Media Access Layer (MAC) and the physical layer that transports the data on the physical medium (electrical signal...)

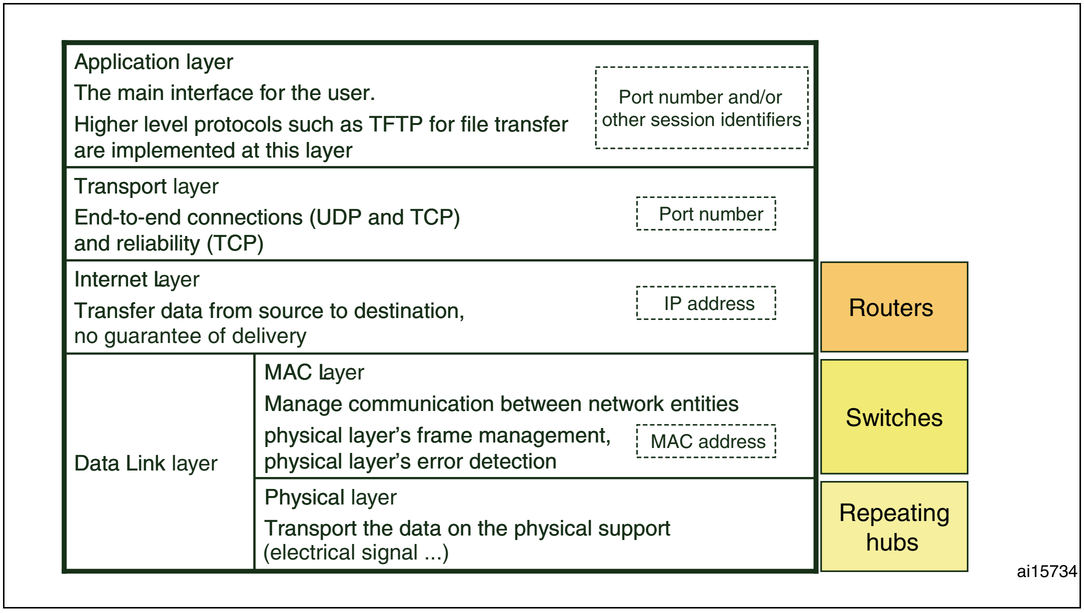
The MAC layer handles the physical layer's error detection and manages the communication between network entities using a dedicated address called the MAC address.

The Link layer is usually fully implemented in hardware, in the STM32F107xx the MAC layer features are implemented in an Ethernet controller embedded in the MCU, while the physical layer is handled by an external chip (PHY).

2.1.5 Example of network devices

This part presents some network devices, starting with devices operating at the link layer, and then giving examples of devices operating at the layers above.

Figure 2. Network devices


Application layer
The main interface for the user.
Higher level protocols such as TFTP for file transfer are implemented at this layer
Port number and/or other session identifiers
Transport layer
End-to-end connections (UDP and TCP) and reliability (TCP)
Port number
Internet layer
Transfer data from source to destination, no guarantee of delivery
IP address
Routers
Data Link layer
MAC layer
Manage communication between network entities
physical layer's frame management, physical layer's error detection
MAC address
Switches
Physical layer
Transport the data on the physical support (electrical signal ...)
Repeating hubs
ai15734

- A repeater hub is a simple device inserted between 2 parts of a network. It operates at the electrical signal level. It forwards all the signals received from one connector to the other connectors. In a half duplex network it manages the collision detection as well. This device is transparent to the network.
- Switches and bridges operate at the data link layer. Switches are used in a single network like repeater Hubs but are more intelligent. A switch is able to use the source and destination MAC Addresses contained in the Ethernet header of a packet to forward it appropriately.

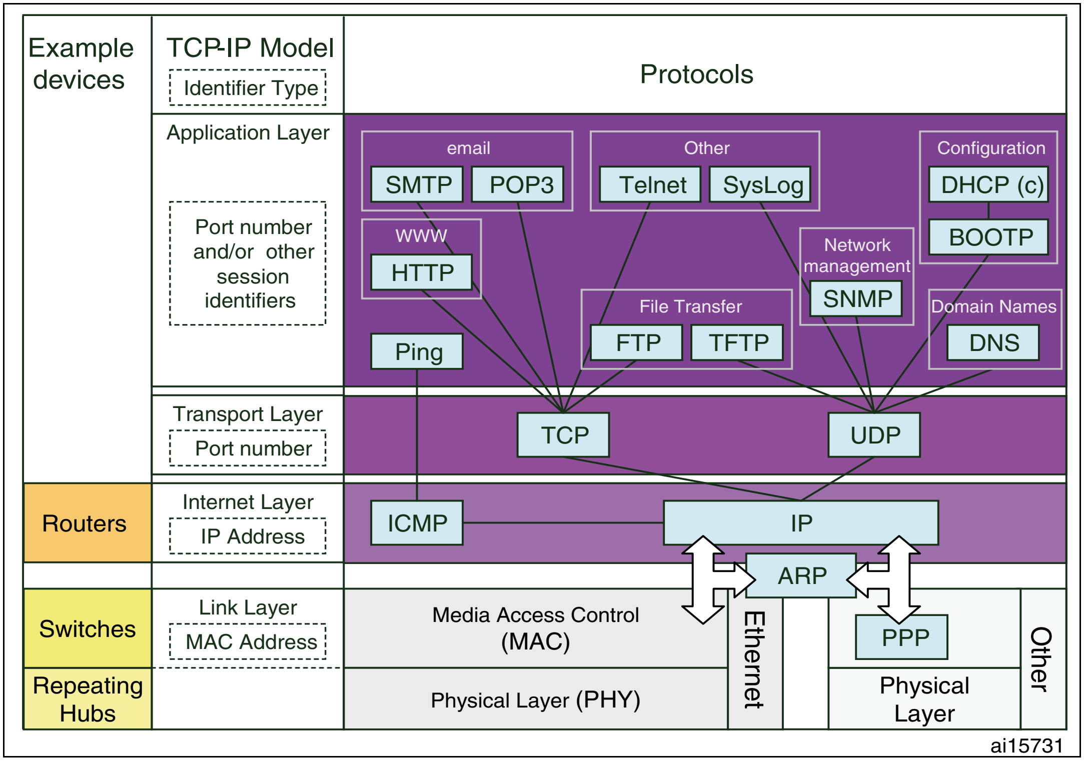
- Contrary to switches and hubs, routers are able to interconnect several networks by operating at the network layer and using the IP addresses of the packets.

These are simple examples, many products operate at different layers. For example “layer 3 switches” are able to perform some router operations by using IP addresses (layer 3 or network layer) while also performing their data link layer operations (also called layer 2 operations).

2.2 The protocols

If we put all this information together and add the main protocols of each layer we get the diagram shown in *Figure 3*.

Figure 3. Protocol interconnections

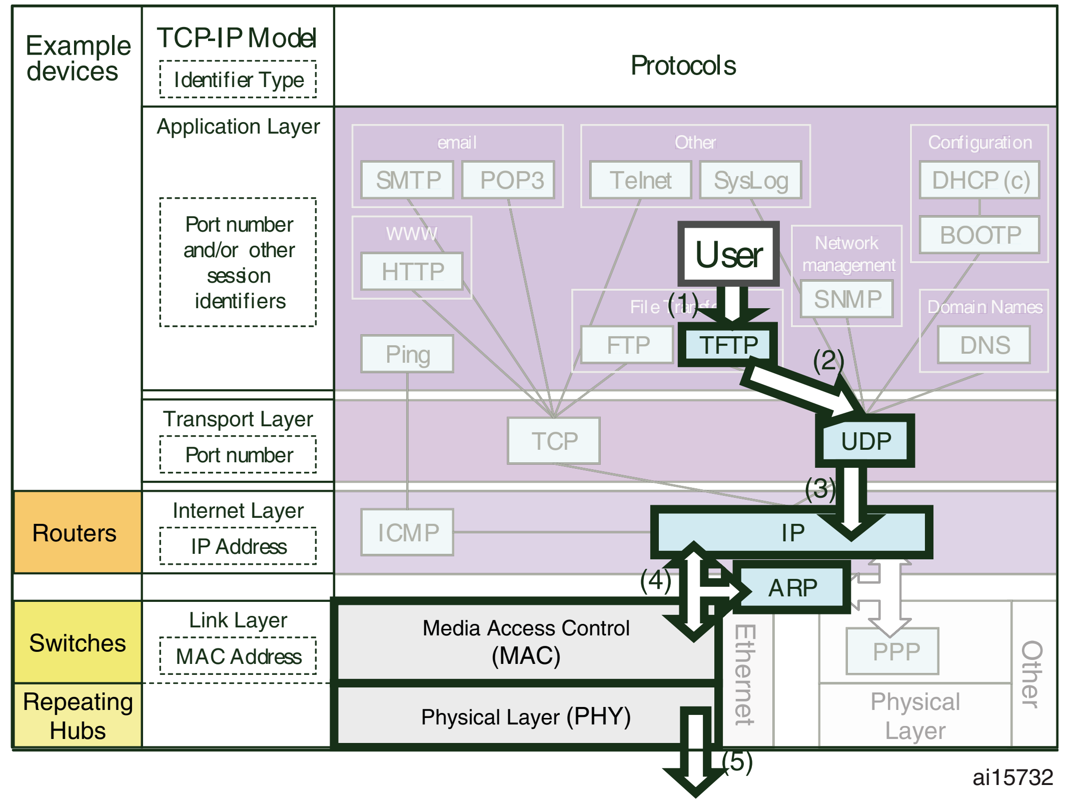


On the right part of *Figure 3* you can see which layer the most common protocols operate at and what the interconnections are between these protocols.

To illustrate this further we can follow the “path” of some user data, a user file for example, to be sent over the network from a device A (MAC Address = AA-AA-AA-AA-AA-AA / IP address = 128.128.128.128) to a device B (MAC Address = BB-BB-BB-BB-BB-BB / IP address = 129.129.129.129). This is shown in *Figure 4*.

Note: Please note that these MAC and IP addresses may be invalid, they are just chosen for this example.

Figure 4. User data path example



Here is a simplified description of the path:

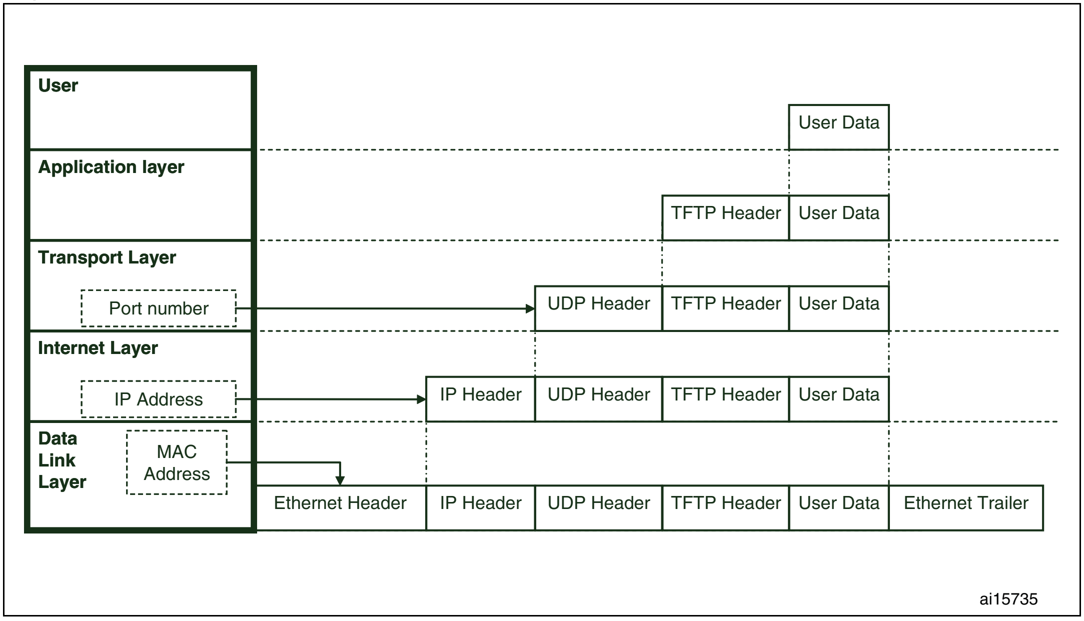
1. The user application uses the TFTP service to send its file.
2. The TFTP uses the UDP service to send a TFTP header and some data, the port used by the UDP is port 69 (reserved for TFTP).
3. The UDP protocol adds a UDP header to the previous data and uses the IP service to send this updated data to device B. The IP address of B is provided to the IP service.
4. The IP protocol appends to the data an IP header containing the source IP address 128.128.128.128 and destination IP address 129.129.129.129 as well as other information. The IP protocol then uses the data link service to send this data to the physical layer.
5. The Data link layer builds an Ethernet frame containing the previously described data and a header containing the MAC address of device A and the MAC address of the destination address B.

BUT, there is a problem here: How does the TCP/IP stack know the MAC address of the destination? Either the MAC address is already known (stored in a table) or the TCP/IP stack has to find out the MAC address from the IP address. This is done using the ARP protocol (Address Resolution Protocol).

Therefore in order to send its data, the TCP/IP stack may have to send some ARP Ethernet frames first to get the MAC address of the destination. Once the stack knows the destination MAC address, it can complete step 5 above.

In this example we can see that each layer adds some information (header) to the data received from the layer above before using the layer below, this is called encapsulation.

Figure 5. Data encapsulation



2.3 Introduction to some TCP/IP protocols

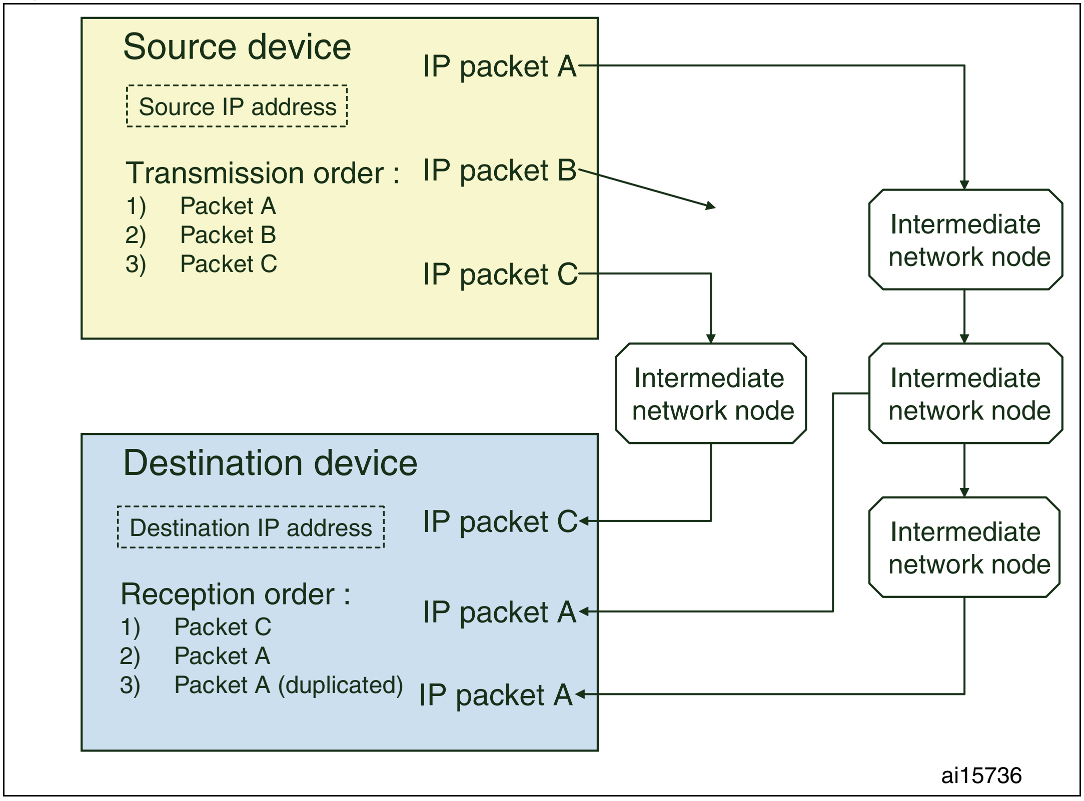
This section briefly describes the different protocols. The introduction order may seem more or less arbitrary as different applications use different protocols, but the intention is to start with a description of the most important protocols before covering the less commonly used ones.

2.3.1 IP: Internet protocol

This protocol is one of the core protocols of the stack, most of the datagrams (TCP, UDP, ICMP, HTTP...) are transmitted using this protocol. IP provides a service for sending data from one identified network entity to another entity that may be located on a distant network. IP addresses are used to identify the source and destination devices.

While allowing data to be sent to devices over the networks, the IP protocol does not guarantee the delivery of this data, IP is a connectionless protocol.

Figure 6. IP protocol transmission and reception



As shown in *Figure 6*, IP datagrams can be lost, duplicated, or received in any order... It is a “best effort delivery” mechanism, the reliability of the communication (retransmission, reordering of the packets) is not handled at this layer.

2.3.2 ARP: address resolution protocol

The ARP is also a key element of the Internet protocol suite. As shown in *Figure 6*, the IP only uses IP addresses to identify the source and destination, but the physical layer requires a MAC address to transmit the data. The Address Resolution Protocol allows an emitting device to get the MAC address of the destination by knowing its IP address. These ARP requests and answers are sent by the stack itself, but the user application sometimes has to be aware of this mechanism (for example, the API may return a code like “ARP request sent”).

2.3.3 ICMP: internet control message protocol

The ICMP protocol allows the communication of error messages and queries between entities over the network; it is actually a very important protocol, although it is less known. ICMP messages are encapsulated in a IP datagram for transmission.

Table 1 gives some examples of queries and error messages that can be sent by ICMP. These messages are acted on by the IP and the higher level protocols.

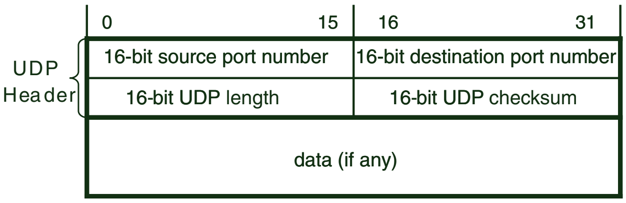
Table 1. ICMP errors and queries

Errors	Queries
Protocol unreachable	Echo request
Port unreachable	Echo reply
TTL (time to live) reaches 0 during transit	....
...	

2.3.4 UDP: user datagram protocol

UDP is a simple datagram service allowing data to be sent over the network, it relies on IP. The UDP packet is very simple; it contains an 8-byte header and the data. The header just adds a source port and destination port number.

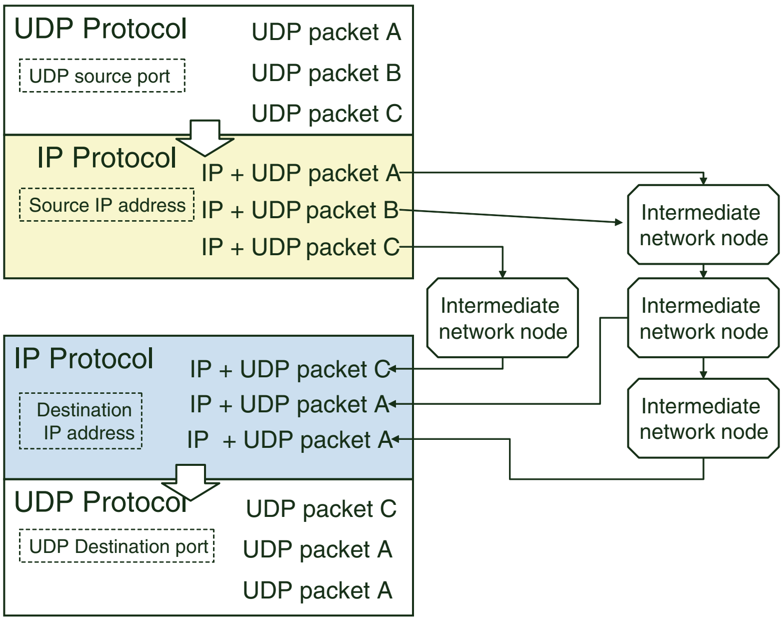
Figure 7. UDP datagram



ai15737

The UDP is a connectionless protocol and has the same drawbacks as the IP layer. It is still used by a lot of protocols and applications. The simplicity of UDP is an advantage for applications where the upper layers already implement mechanisms that ensure the reliability of the communication, or for protocols that do not require this reliability.

Figure 8. UDP protocol transmission and reception

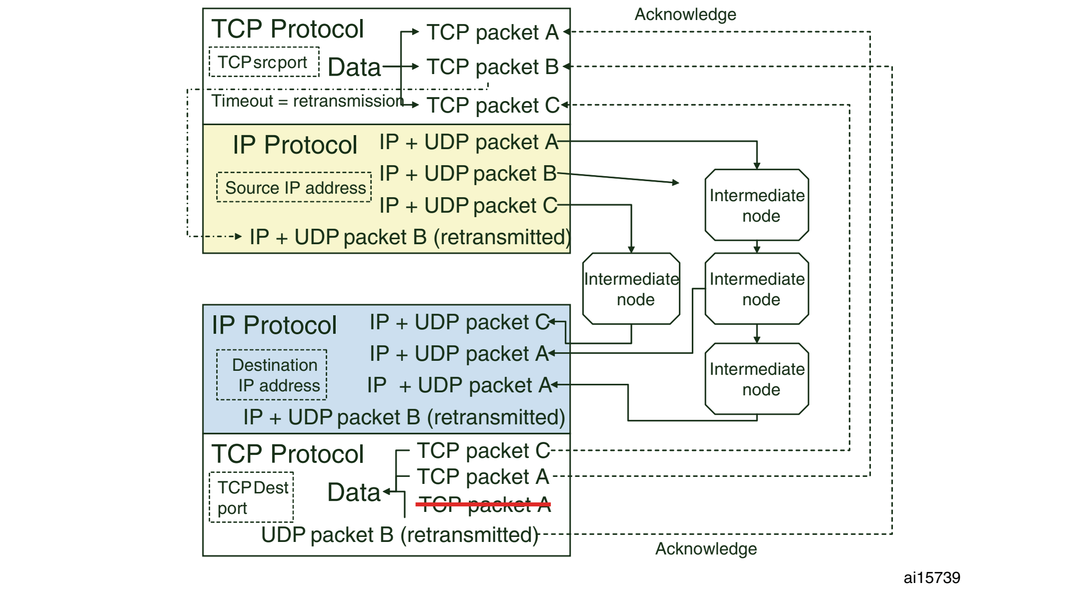


ai15738

2.3.5 TCP: transmission control protocol

The transmission protocol is one of the key elements of the Internet Protocol Suite. Unlike the IP and UDP protocols it provides a "connection oriented" service, allowing reliable communication between 2 devices on different networks. TCP relies on the IP service for the transmission of the data over the networks.

Figure 9. TCP protocol transmission and reception



This reliability is achieved with the following mechanisms implemented in the TCP:

- A checksum to check the validity of the datagram like other protocols
- An acknowledge mechanism for the receiver to confirm to the sender the proper reception of a packet. If the sender does not receive this acknowledge after a timeout it resends the packet.
- If needed, the data to send over TCP is split in segments, each segment being transmitted in a separate TCP packet, the receiver has a re-sequencing mechanism to reorder packets if they are received out of order.
- Duplicated packets are identified and discarded by the TCP protocol
- A TCP window mechanism to allow a device to specify the buffer size available and prevent another device from sending more data than it can handle.

2.3.6 BOOTP: bootstrap protocol

The Bootstrap protocol allows a device to get an IP address from a configuration server. It relies on the UDP service to operate. This protocol is embedded, for example, in the BIOS of some network devices to allow these devices to boot from the network.

2.3.7 DHCP: dynamic host configuration protocol

DHCP also offers some of the services of the BOOTP protocol. It allows a device to get an IP address from a configuration server. Newer than the bootstrap protocol, the DHCP has more advanced features. Many devices now implement the DHCP client to retrieve an IP address, fewer device implement the server service that allocates the IP addresses. One of the advanced features of DHCP over BOOTP is the ability to allocate an IP address to a client for a limited time (the BOOTP allocation being unlimited). The DHCP supports the Bootstrap protocol.

2.3.8 TFTP: trivial file transfer protocol

First defined in 1980, the TFTP protocol has a client/server architecture and allows file transfer over UDP using a simple mechanism allowing easy implementation in memory constrained devices.

To overcome the limitation of UDP that does not guarantee packet delivery, the receiver sends an acknowledge packet after reception of a packet. Only one packet (either data or acknowledge) is sent over the network at a time, reducing the performance compared to FTP protocol for example.

To summarize, the main advantage of TFTP is its simplicity, the drawback being its performance on high latency networks. This protocol is used for example to download boot loaders.

2.3.9 FTP: file transfer protocol

FTP allows file transfer via the TCP protocol, and therefore it inherits its connection oriented characteristics. It also takes advantage of the window mechanism and reordering abilities of the TCP protocol, so that several packets can be sent over the network simultaneously increasing the performance. This client-server protocol uses 2 ports, one for data, the other one for control information.

FTP supports password based connections or anonymous users. Please note that FTP doesn't include an encryption mechanism, so the user name and password are sent over the network in clear text.

2.3.10 PPP: point-to-point protocol

The point-to-point protocol (PPP) provides a standard method of transporting multiprotocol datagrams over point-to-point links. It is used over many types of physical networks including serial lines and Ethernet.

2.3.11 SMTP: simple mail transfer protocol

The SMTP is a standard protocol used to transmit electronic mails across networks. It relies on the connection oriented TCP protocol. Like most network protocols it has a client/server architecture. The commands are coded in ASCII, making the frames easier to understand. Each command from the client is followed by an answer from the server.

Table 2 lists the commands supported by SMTP:

Table 2. SMTP commands

Command	Description
HELO or EHLO	Commands used by the client to identify itself
MAIL FROM:	Identifies the originator of the message
RCPT TO:	Identifies the recipient of the message
DATA	Command used to send the content of the message
QUIT	Terminate the email exchange

SMTP is a protocol for transmitting emails between devices that are always connected to the networks. It has no features for retrieving messages from a server (see POP3)

2.3.12 POP3: post office protocol version 3

POP3 (Post Office Protocol version 3) is a protocol designed to retrieve emails from an account (mailbox) on a server (post office). It has no features for sending messages to servers (see SMTP).

2.3.13 SNMP: simple network management protocol

This protocol relies on UDP to offer a coherent framework for managing networks. This is necessary, as networks are growing in size, complexity and diversity. It is a client /server (also called agent/manager) architecture.

This protocol has a 2 way communication scheme, the manager being able to ask the agent for some information, the agent also being able report events.

2.3.14 HTTP: hypertext transfer protocol

HTTP is the protocol used by the World Wide Web; it has client/server architecture and uses a request/response mechanism to operate.

Once the connection is opened, the HTTP client (usually your browser) sends a request message to an HTTP server. The server returns a response message that usually contains the requested resource (an html file for example). The connection is then closed. The HTTP protocol does not maintain the connection between transactions.

2.3.15 Telnet: remote terminal protocol

The Telnet protocol gets its name from Telecommunication network. The purpose of this protocol is to provide a bidirectional communication service on the Internet or on local area networks. Telnet can be used to access a remote host with a terminal. Telnet relies on the connection oriented TCP protocol.

3 NicheLite TCP/IP stack overview

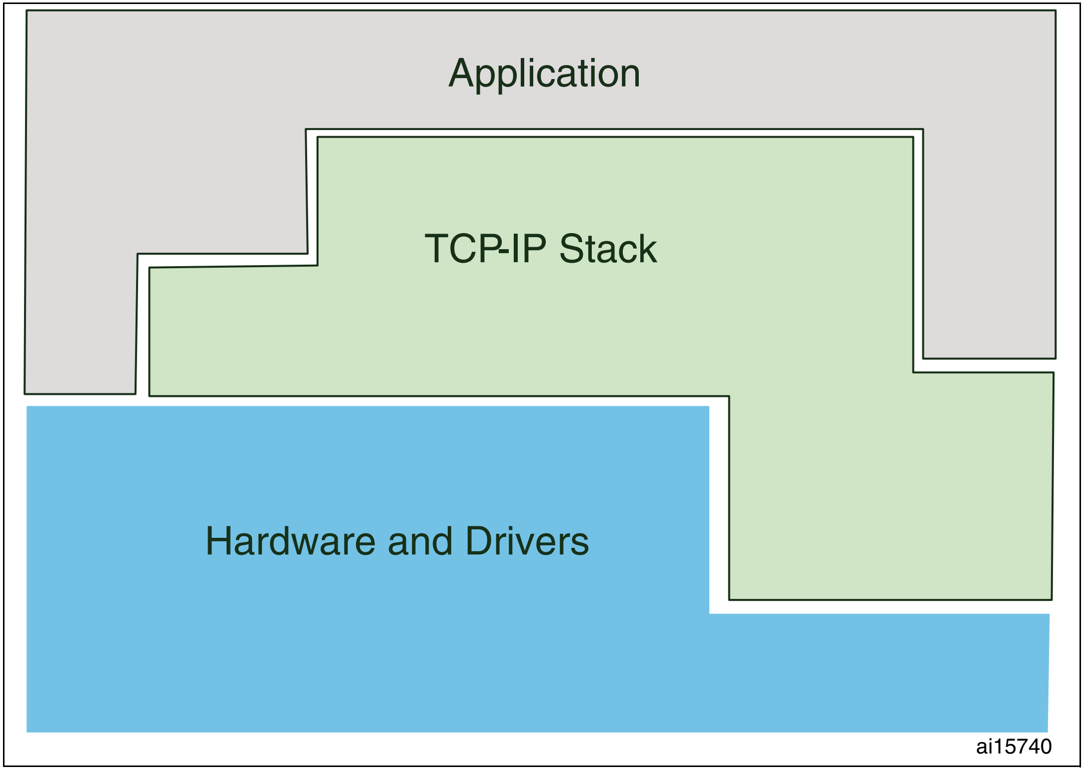
This section provides an overview of the NicheLite TCP/IP stack. *Section 3.1* introduces the structure of the stack. *Section 3.2* describes the protocols available in the stack. *Section 3.1* gives an introduction to the NicheLite stack implementation including packet demultiplexing and buffer management.

3.1 Stack structure

As shown in *Figure 10*, a connectivity application can usually be divided in several parts:

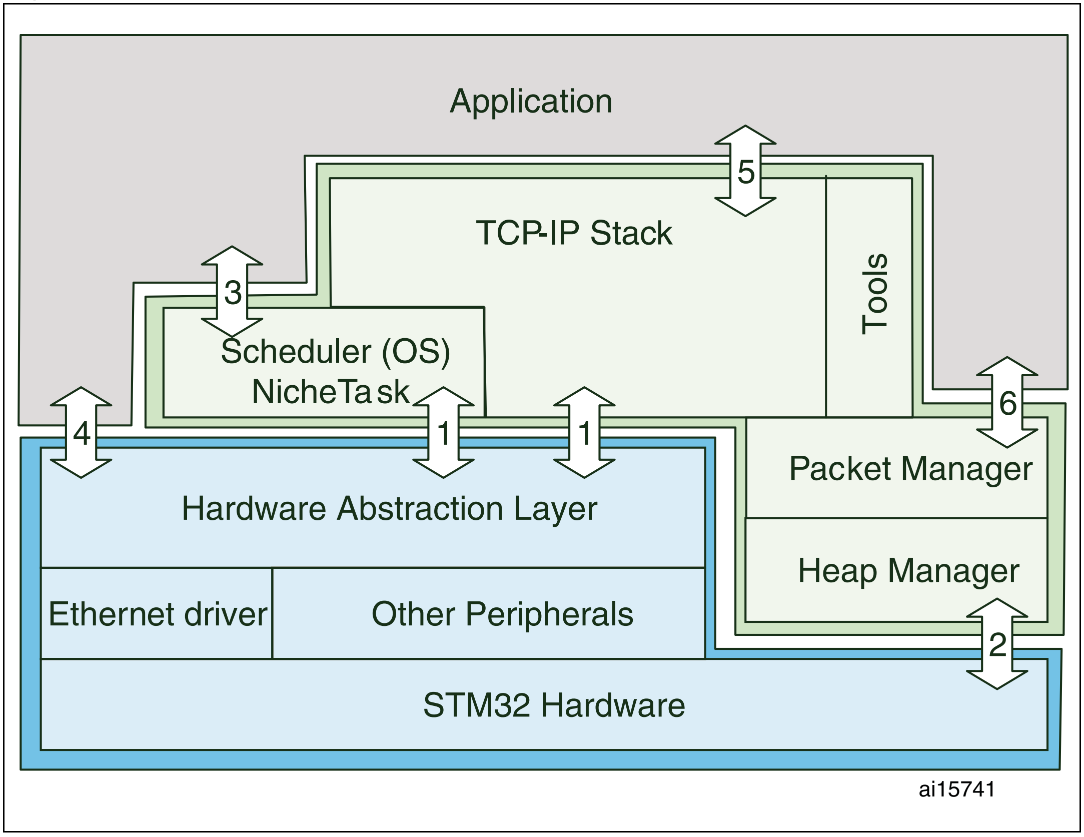
- The application layers
- The TCP/IP stack layers
- The hardware abstraction layer, mainly containing the low level driver

Figure 10. Connectivity application layers



A more detailed diagram is shown in *Figure 11*.

Figure 11. Stack components



In this diagram the TCP/IP layer has several parts:

- a simple task scheduler (called NicheTask),
- a part that is dedicated to the memory management (Packet and heap managers)
- TCP/IP core part handling the main protocols.

Note: 1 *The TCP IP layer and NicheTask mainly rely on the hardware abstraction layer to access the STM32 peripherals.*

2 *The stack memory management modules directly access the RAM of the MCU to manage the buffers.*

3 *You can use the NicheTask operating system to add specific tasks for your application.*

4 *The application can access the STM32 standard peripherals (Timers, GPIO, SPI, USART…) using the STM32 Standard Peripherals Library.*

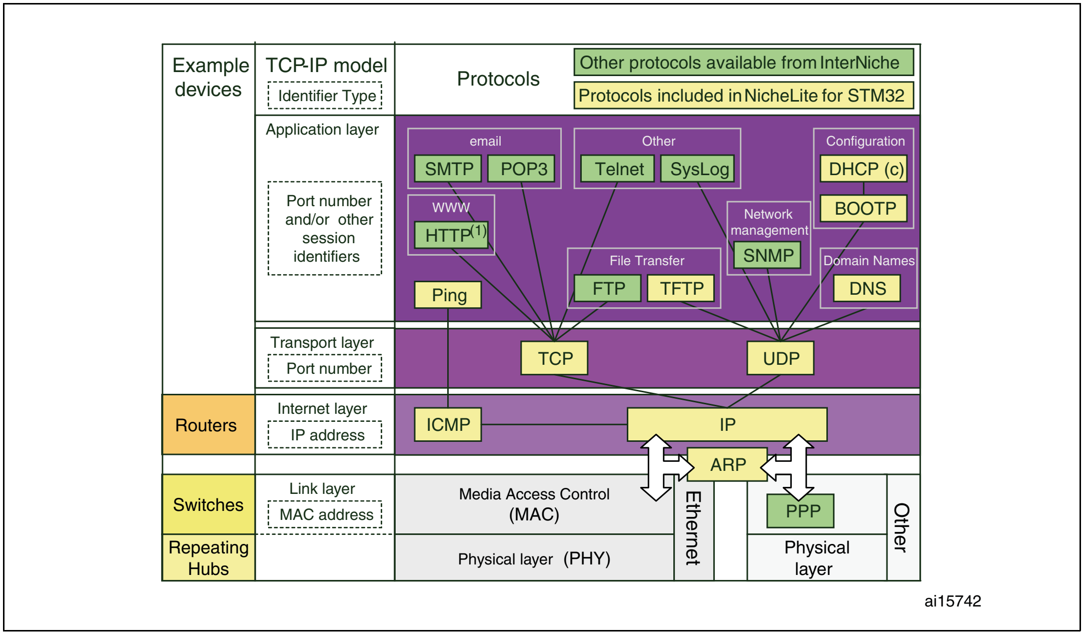
5 *The user application relies on the TCP/IP stack to implement its network services.*

6 *An important part of the TCP/IP configuration requires the developer to specify some memory management related parameters (such a buffer size, buffer numbers…).*

3.2 NicheLite protocols

Figure 12 shows the protocols available in the TCP/IP stack from InterNiche. Please note that some protocols are embedded in the free NicheLite for STM32 package (yellow boxes), while other protocols are available from InterNiche (green boxes).

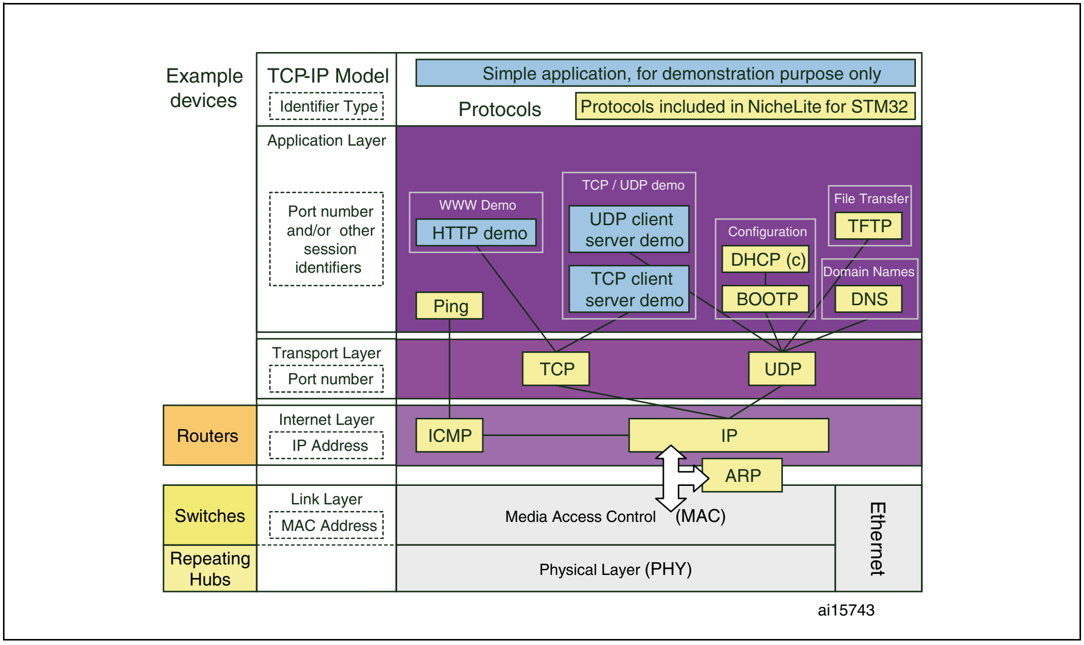
Figure 12. Protocols available in NicheLite stack



1. A simple HTTP demo based on NicheLite is available from ST.

Figure 13 shows the protocols and demos available in the STM32 free package.

Figure 13. Demos available in STM32 package



Here are the characteristics of this package:

- Demos
  - Simple HTTP server demo (no CGI)
  - Simple UDP client-server demo
  - Simple TCP client/server demo
  - Simple TFTP client/server demo
- Application layer
  - Dynamic Host Configuration Protocol (DHCP) Client
  - Name Service (DNS) Client
  - Bootstrap Protocol (BOOTP)
  - Trivial File Transfer Protocol (TFTP), client and server
  - Ping
- Transport layer
  - Transmission Control Protocol (TCP) supporting InterNiche's lightweight API, and a Zero-Copy option
  - User Datagram Protocol (UDP)
- Internet layer
  - Internet Protocol (IPv4), without fragmentation and reassembly
  - Internet Control Message Protocol (ICMP)
  - Address Resolution Protocol (ARP)

The following other modules and features are provided in the package as well:

NicheTool

NicheTool is a UART user interface, that allows commands to be sent using a HyperTerminal window. The commands already implemented in NicheLite are useful for performing some TCP/IP stack diagnostics. NicheLite provides APIs that allow you to easily add your own commands.

NicheFile virtual file system

NicheFile is a virtual file system used by the TFTP protocol (and other modules from InterNiche like the optional HTTP web server). The TFTP client (or server) implementation uses this virtual file system to present a file system architecture to the server (or client). But there may not be a real file system implemented in the device for optimization reasons.

IPv4 without fragmentation

NicheLite only supports IPv4 without fragmentation and reassembly. This means that IPv6 datagrams are discarded, and the fragmentation / reassembly that is usually part of the IP layer has to be handled by the upper layers.

ZeroCopy

NicheLite implements a zero copy mechanism, this means that the encapsulation described in *Section 2.2* is done without any recopy when the data are passed from one layer to the other. This increases the performance.

Minisocket API

NicheLite also implement a mini-socket API that differs from the BSD socket API used by some other implementations. The mini-socket API is designed to be as close as possible to the BSD socket API while still allowing a small footprint. The primary differences are that passive connections are accomplished with a single call, m_listen(), rather than the BSD bind()-listen()-accept() sequence. The BSD select() call is also replaced with a callback mechanism.

BSD = Berkeley Software Distribution TCP/UDP/IP Stack IP

Table 3. Mini-socket and BSD socket APIs

Mini-Sockets	BSD Sockets
m_socket()	socket()
m_connect()	connect()
m_recv() and/or m_send() - or - tcp_send() and/or tcp_recv() - (zero-copy I/O)	recv() and/or send()
m_close()	close()

3.3 NicheLite implementation introduction

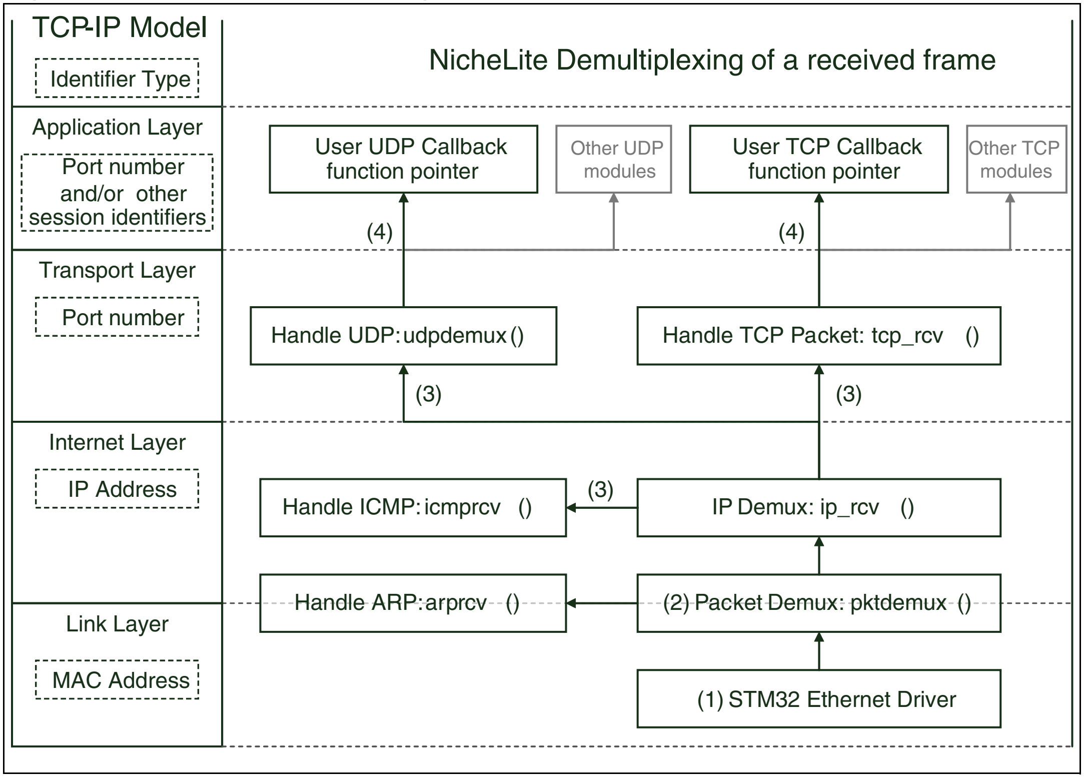
This section presents the NicheLite implementation:

- First from a execution flow point of view: *Section 3.3.1* describes the path followed by data from the time it is received by the Ethernet peripheral until it reaches the user application
- Then from a memory point of view: *Section 3.3.2* introduces the stack memory management.

3.3.1 Packet demultiplexing flow

Figure 14 shows the demultiplexing flow of data received by the Ethernet peripheral.

Figure 14. Packet demultiplexing



Here is the sequence:

1. The data is received by the Ethernet peripheral and handled by the Ethernet driver.
2. Then when the system is available to run the TCP/IP task, the function pktdemux() is called. This function detects if the Ethernet frame received is an ARP datagram or an IP datagram, and calls arprcv() or iprcv() accordingly.
3. The iprcv() function checks if the datagram is an IPv4 datagram, checks the validity of the IP frame (checksum, length….) and then calls icmprcv(), udpdemux(), or tcp_rcv() depending on the datagram protocol
4. If the user application has already set up a listener to “listen” for the packet on a UDP or TCP socket, and if a datagram for this socket is received then the corresponding callback function is called and used to provide this packet to the user application

Figure 14 briefly describes the flow of a packet. The stack implements more functions and verifications than are described in this introduction. The memory management is presented in the next section.

3.3.2 Packet buffer memory management

NicheLite uses 2 types of packet buffer, little and big buffers, depending on the context, the stack uses either little or big buffers. This dual buffer mechanism gives you better control of the RAM footprint depending on the characteristics of your application and your network.

Here is some additional information about these buffers and the behavior of the stack:

- The size of the big buffers depends on the MTU (maximum transmission unit). Refer to *Section 5: NicheLite TCP/IP stack optimization* for more information.
- The size of the little buffers is defined by the developer.
- The reception of a packet always uses a big buffer. In fact, the device doesn't know the size of the packet before receiving it.
- The transmission of a packet uses a little or a big buffer depending on the size of the packet to be transmitted.

This is further discussed in *Section 4*.

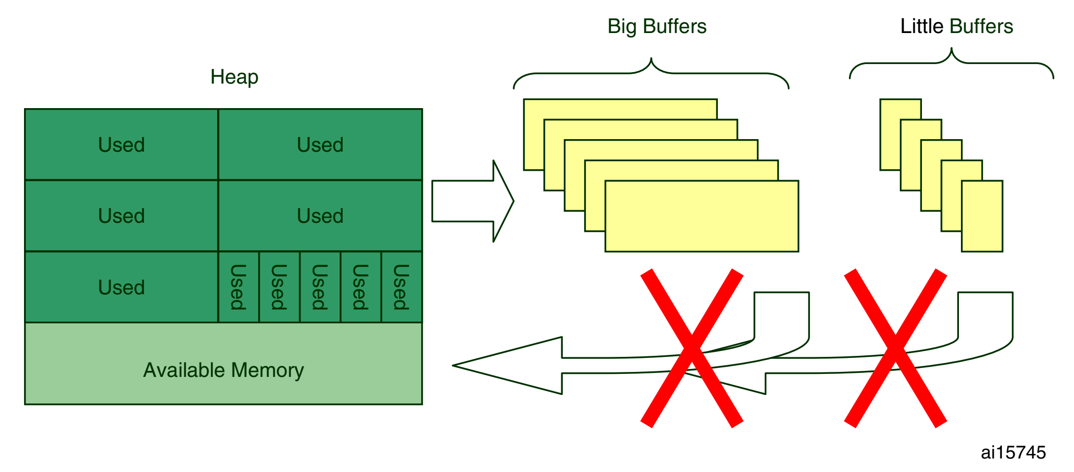
Step by step introduction to the buffer management

The *Figure 15* to *Figure 20* provide a pictorial introduction to the packet buffer management

1. During its initialization the TCP/IP stack allocates the number of little and big buffers defined by the developer.

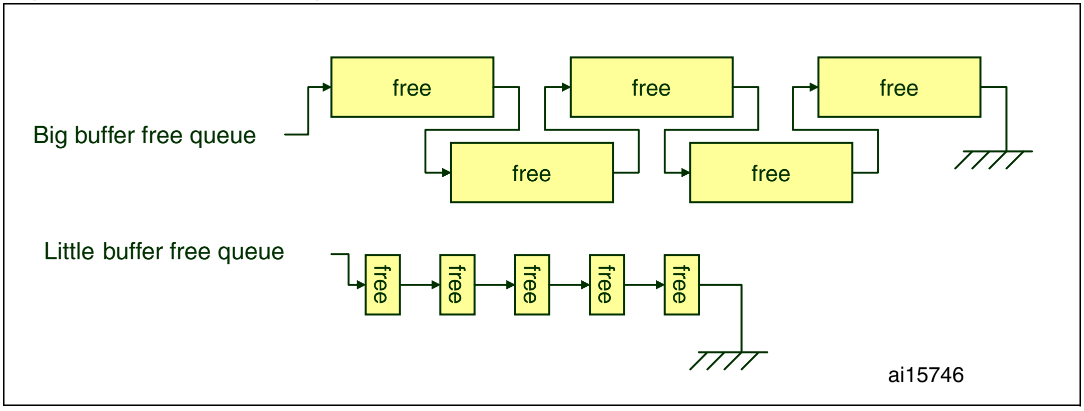
   These buffers are managed by the stack, and never returned to the Heap.

Figure 15. Buffer management (1)



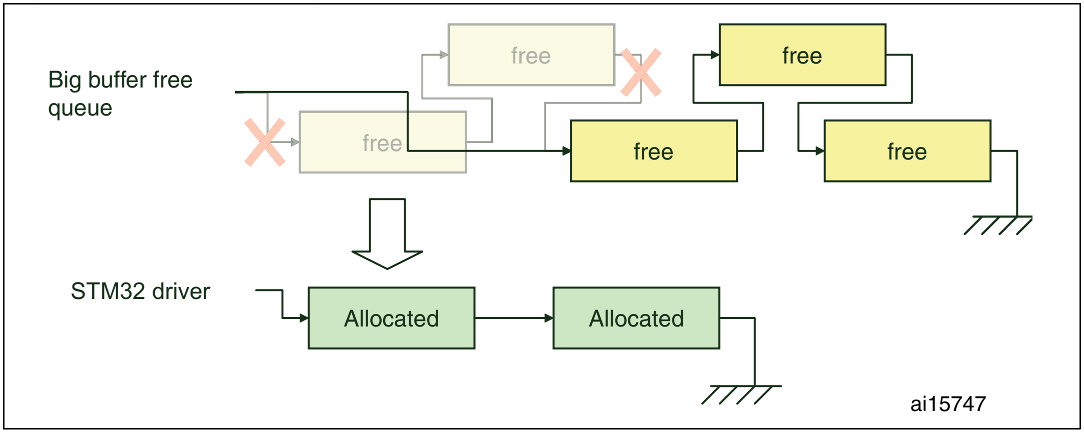
2. Once allocated from the Heap, the buffers are organized in 2 linked lists:
   – Big buffer free queue
   – Little buffer free queue

Figure 16. Buffer management (2)



3. During the initialization, the Ethernet driver allocates 2 big buffers from the free queue. These buffers are used to store the frame are received by the Ethernet peripheral.

Figure 17. Buffer management (3)

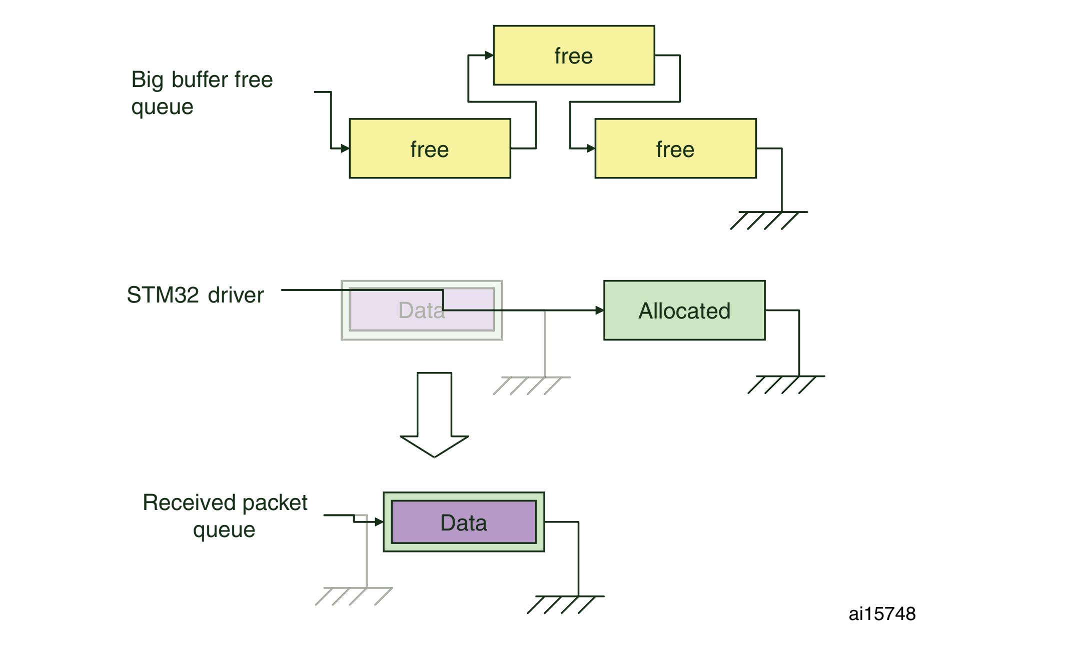


4. Upon reception of a frame, the STM32 Ethernet driver copies the data received by the Ethernet peripheral into one of the 2 previously allocated buffers.

   Then this buffer is moved to another queue: the received packet queue. The received packets are stored in this queue until the stack is able to process it.

   This received packet queue can be seen as a kind of cache that allows the reception mechanism and the TCP/IP stack to act as 2 separate tasks.

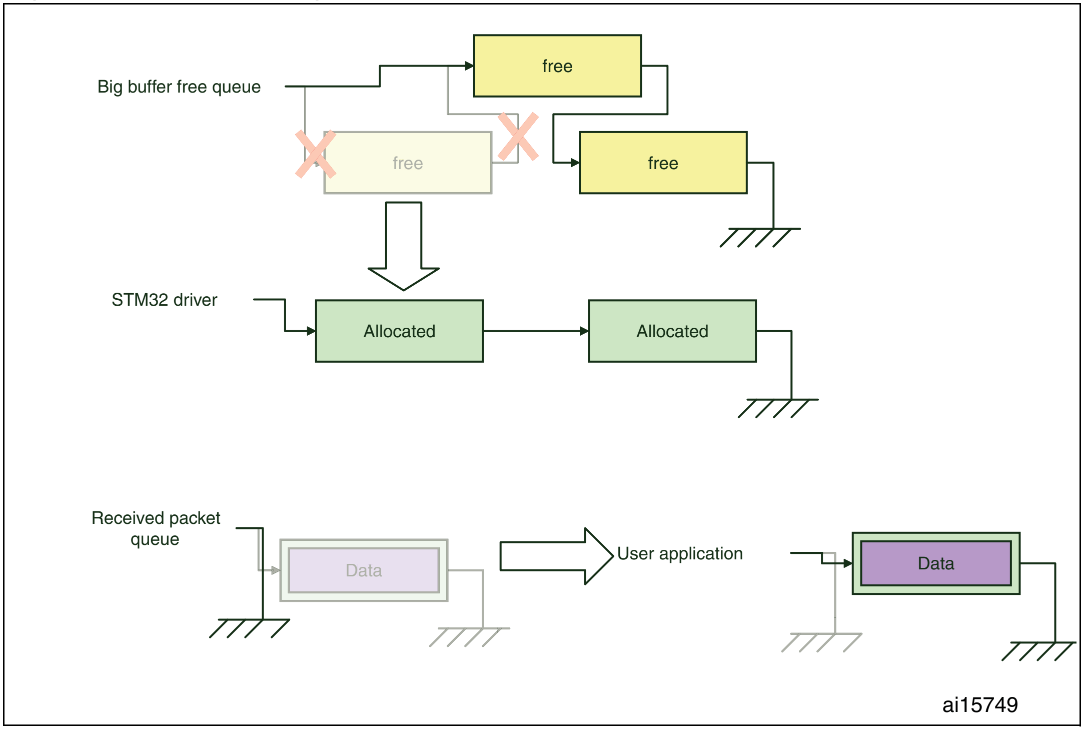
Figure 18. Buffer management (4)



5. Then the driver reallocates a buffer from the big buffer free queue.

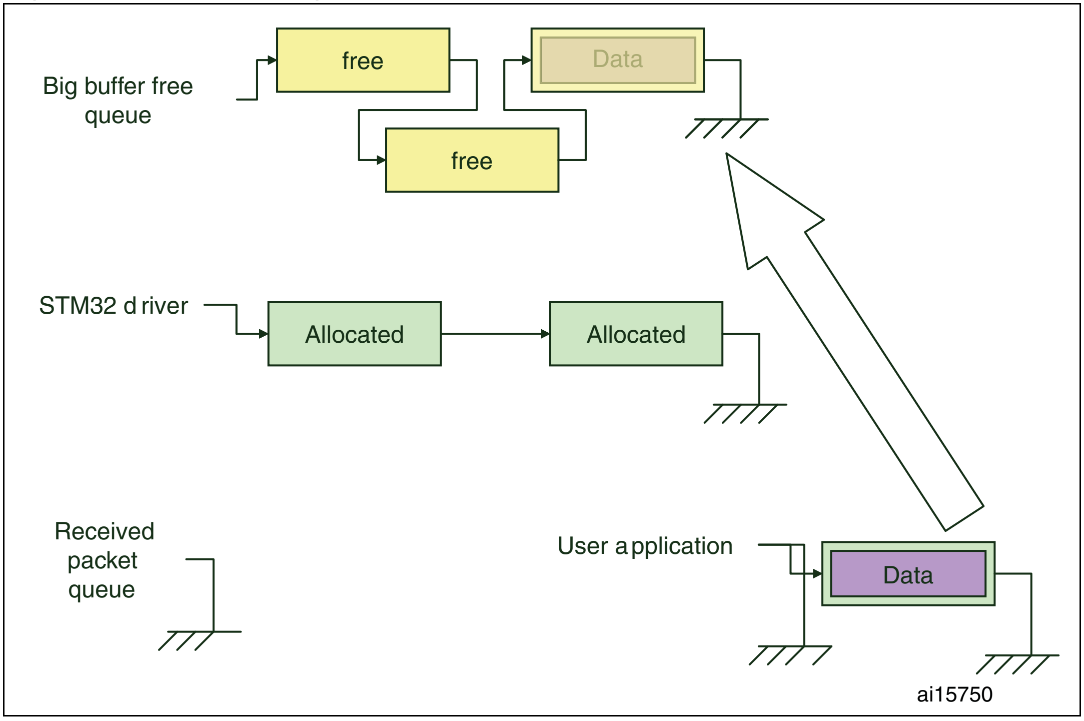
   The TCP/IP stack handles the received packet, as described in *Section 3.3.1*. If a UDP or TCP packet is received and if the corresponding UDP or TCP listener exists, then the buffer pointer is provided to the user application.

Figure 19. Buffer management (5)



6. Once the packet is handled by the user, it has to be given back to the big buffer queue. Please note that for performance reasons, the data is not erased.

Figure 20. Buffer management (6)



3.3.3 Memory management

Memory management is critical for a TCP/IP based application to work properly. You need to carefully evaluate your application requirements in terms of memory and test it thoroughly.

Several listeners

An application can have several listeners for example, one listener for a UDP-based application and another for a TCP-based application (see *Figure 21*).

This context has to be identified and checked carefully.

For example, in *Figure 21*, the 2 listeners (UDP and TCP) are using several buffers and there is no big buffer left in the big buffer free queue.

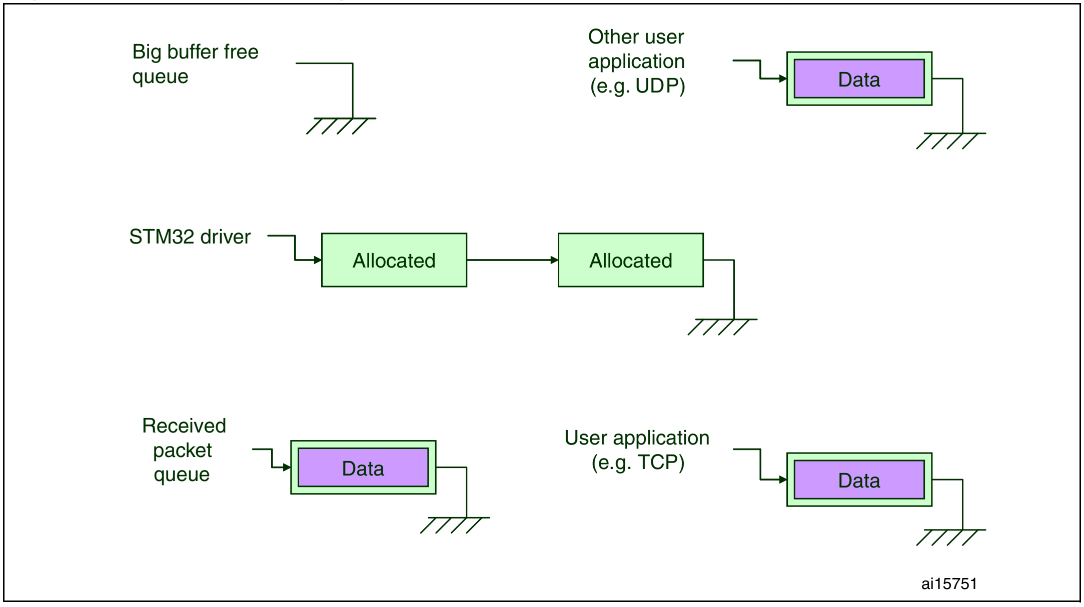
This situation is serious but the application is still able to:

- receive 2 packets (there are still 2 buffers allocated in the driver)
- send out some data if they fit in a small buffer (and if there is a small buffer available in the small buffer free queue)

This will become a critical situation if:

- 2 more buffers are received and the UDP or TCP application does not release the buffers
- the application needs to send some data that do not fit in a small buffer, and there is no way of allocating a big buffer to this sending operation

Figure 21. Memory management: a serious situation that may become critical

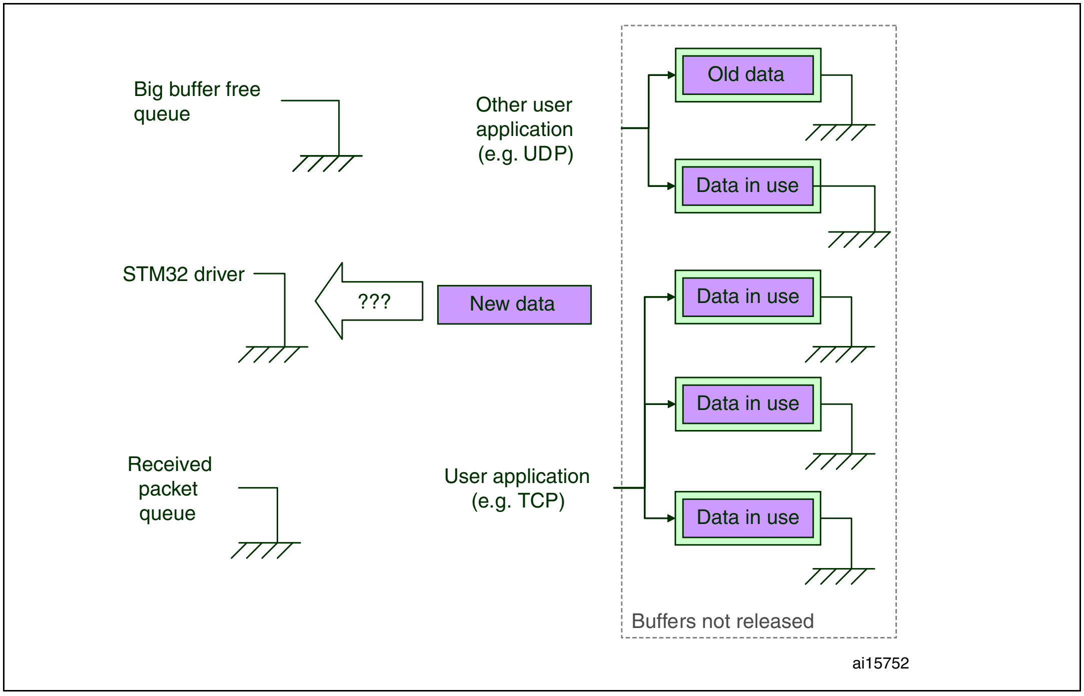


No free buffers

If the buffer requirements of an application are underevaluated, or if the buffers are not returned to the free queue, then we may be in this context.

In this example, all the buffers are kept by the application, the next data received are lost as there is no buffer left at driver level (see *Figure 22*) and the application is not able to send any data (unless they fit in a small buffer).

Figure 22. Memory management: a critical situation



4 NicheLite TCP/IP stack configuration

4.1 NicheLite modules

NicheLite stack components can be enabled or disabled depending on the application requirements. This is done through the ipport.h header file. A set of preprocessor defines are available to select components such as NicheTask scheduler, DHCP, DNS… (Modules not supported by NicheLite should not be enabled).

The next sections provide a description of how to use some of the defines available in ipport.h file.

4.2 IP address

The application could use a fixed IP address (static) each time it powers up, or it can get an IP address assigned automatically by a server using the Dynamic Host Configuration Protocol (DHCP).

The configuration of the IP address is performed in the function pre_task_setup() in the in_stubs.c file.

Here is an example of an implementation taken from the STM32 demo:

```
#ifdef DHCP_CLIENT
  netstatic[0].n_flags |= NF_DHCPC;
  netstatic[i].n_ipaddr = htonl(0x0a0000c5);   /* 10.0.0.197 */
#else
  netstatic[i].n_ipaddr = htonl(0x0a0000c5);   /* 10.0.0.197 */
#endif
  netstatic[i].snmask  = htonl(0xffffff00);    /* 255.255.255.0 */
  netstatic[i].n_defgw = htonl(0x0a00008A);    /* 10.0.0.138 */
  i++;
```

When using a static IP address you can modify the default address defined by the n_ipaddr field:

```
#else
  netstatic[i].n_ipaddr = htonl(0x0a0000c5);   /* 10.0.0.197 */
#endif
```

When DHCP is enabled (DHCP_CLIENT is set to 1 in ipport.h file), the n_ipaddr field is ignored and stack initialization finishes only after the DHCP transaction is completed and a dynamic IP address is acquired. Otherwise if the DHCP fails the static IP address is assigned.

4.3 MAC address

The default MAC address for the STM32F107xx Ethernet controller being used is defined in the mac[] array, which is initialized in the stm32.c file. The address array is filled using the ETH_MACAddressConfig() function.

Here is an example of an implementation taken from the STM32 demo:

Note: If you need to load this demo on different STM32 devices and connect them on a public network, you have to change the MAC address accordingly to have unique address for each device (for example for each device change the value of MAC_ADDR5)

```
/* STM3210C-EVAL Board MAC Address Fixed */

#define MAC_ADDR0 0x00

#define MAC_ADDR1 0x53

#define MAC_ADDR2 0x54

#define MAC_ADDR3 0x91

#define MAC_ADDR4 0x00

#define MAC_ADDR5 0x01

/* Mac address sent in the Ethernet frame */

char mac[] = {MAC_ADDR0, MAC_ADDR1, MAC_ADDR2, MAC_ADDR3, MAC_ADDR4, 
MAC_ADDR5};

...

/* MAC address configuration */

ETH_MACAddressConfig(ETH_MAC_Address0, (u8*)mac);
```

Given that the MAC address must be unique for every end-user Ethernet application in the world and since it is not possible to change the mac[] array and compile the code for each application's product; you have to implement a way to program the MAC address in non-volatile memory and make it available to the stack during the initialization phase.

4.4 Keepalive

A keepalive is a message sent by one device to another to check that the link between the two is operating. The message is short and sent at programmable intervals so as not to take too much bandwidth on the network.

4.4.1 Enabling Keepalive in NicheLite

The keepalive mechanism should be enabled in server applications that might otherwise hang indefinitely and consume resources unnecessarily if a server or client crashes or aborts a connection during a network failure.

But do remember that it generates extra network traffic, which can have an impact on routers and firewalls.

When creating a new socket you can enable the keepalive mechanism by setting the value of so_options, member of the socket structure, to SO_KEEPALIVE.

Here is an example of an implementation taken from the STM32 Client demo:

```
WP_SOCKTYPE client_sock = INVALID_SOCKET;

...

client_sock = m_socket(); /* Allocates a socket structure for an
active connection */

client_sock->so_options |= SO_KEEPALIVE;

...

m_connect(client_sock, &client_sin, client_upcall); /* Starts an
active connection */
```

4.4.2 Keepalive timeout tuning in NicheLite

Keepalive timeout is defined by the three parameters:

- **Keepalive time** is the time of connection inactivity after which the first keepalive request is sent. This parameter is defined by the TCPTV_KEEP_IDLE variable in NicheLite.
- **Keepalive Interval** is the duration between two successive keepalive retransmissions (also known as probe), if acknowledgement to the previous keepalive transmission is not received. This parameter is defined by the TCPTV_KEEPINTVL variable in NicheLite.
- **Keepalive Retry** is the number of keepalive requests retransmitted before the connection is considered broken. This parameter is defined by the TCPTV_KEEPCNT variable in NicheLite.

Knowing these parameters we compute the Keepalive timeout value as follows:

timeout = TCPTV_KEEP_IDLE + (TCPTV_KEEPINTVL * TCPTV_KEEPCNT)

By default these parameters are defined in the mtcp.h file as follows:

```
#define  TCPTV_KEEP_IDLE   (120*60*PR_SLOWHZ)

#define  TCPTV_KEEPINTVL   (75*PR_SLOWHZ)

#define  TCPTV_KEEPCNT     8
```

With these values, the default keepalive timeout (time after which the connection with the remote is dropped) is set to 130 minutes which may be too long for an embedded system.

Modifying this value is straightforward: you need only to change these define values with the new ones. For example if you need to configure your application so that keepalive starts after one minute of connection inactivity with 6 probes, 10 seconds gap between each. Your application detects a dead connection after 2 minutes:

```
#define  TCPTV_KEEP_IDLE   (60*PR_SLOWHZ)

#define  TCPTV_KEEPINTVL   (10*PR_SLOWHZ)

#define  TCPTV_KEEPCNT     6
```

5 NicheLite TCP/IP stack optimization

In the TCP/IP stack, the exchanged data (also known as payload) are managed as packets which are stored in the device internal RAM memory.

When the TCP/IP stack execution starts, all packet buffers are allocated in the Heap memory (located in RAM) for the entire application run time.

All packets are placed in a pool of free packets (free queue) after allocation. When a packet is used by the stack it is moved out of this pool and back into it when it is no longer needed by the stack. The management of the packet buffer queue where packet data are stored is performed in the pktalloc file.

For more details on buffer management, refer to *Section 3.3.2: Packet buffer memory management on page 23*.

These packet buffers are used for the management of the stack's little and big buffers. The number of these buffers depends mainly on:

- Ethernet frame size (max payload exchanged on the network)
- Application performance and RAM size requirements (which define the Heap memory size)

Ethernet frame size

The Ethernet frame size is defined by the ETH_MAX_PACKET_SIZE variable in the stm32_eth.h file and it depends mainly on the maximum packet size that can be received or transmitted on the network, defined by MAX_ETH_PAYLOAD variable.

```
#define ETH_MAX_PACKET_SIZE    ((ETH_HEADER) + (ETH_EXTRA) +
                                 (MAX_ETH_PAYLOAD) + (ETH_CRC))
```

By default, MAX_ETH_PAYLOAD is set to 1500 bytes,

```
#define MAX_ETH_PAYLOAD    1500 /*Maximum Ethernet payload size */
```

but this value can be tailored depending on the network requirements.

For example: a DHCP request is 592 bytes long max, if you know there is no data bigger than DHCP on the network, you can set payload to 600.

As NicheLite IP Layer doesn't fragment data, you have to take care that NO received data frame is bigger than 600.

For information, the IP layer in the FULL version of the NicheLite stack fragments data if needed.

Once the Ethernet frame size is fixed, the size of the big buffer is defined as follows:

```
#define  BIGBUFSIZE     ((ETH_MAX_PACKET_SIZE) + (ETHHDR_BIAS))
```

The size of the little buffer is fixed at 200 bytes:

```
#define  LILBUFSIZE     200
```

By default we compute the same number of big and little buffers in order to takes 85% of the HEAP size. The remaining 15% of memory can be used for other allocations like the Virtual File System for example. This is user modifiable.

```
#define  NUMLILBUFS     NUMBIGBUFS
#define  NUMBIGBUFS     ((((HEAP_SIZE)*85)/100)/(((BIGBUFSIZE)+
                            sizeof(struct netbuf))+((LILBUFSIZE)+
                            sizeof(struct netbuf))))
```

However you can modify the number of big and little buffers based on the application requirements and the characteristics of the network.

Here are some cases to be considered:

- In an application that is mainly sending little packets, increasing the number of little buffers and decreasing the number of big buffers can be a good option.
- If the network traffic is mainly little packets, reducing the Ethernet frame size (i.e. size of big buffer) may be another option as well.
- If connected on a public network you may have a lot of broadcast packets sent by servers or other devices, these packets add an extra load to your application. In this case the number of big buffers should be adjusted accordingly.
- It can be desirable to increase the number of buffers available in an embedded system to let the application working perfectly even in the cases where the device receives more packets than expected in an upgraded network. You should make sure to have free buffers available during the application execution, to avoid situations when all buffers are allocated and a need for a new buffer occurs. In this case the new packet is discarded which may impact the stability of the application.

Heap allocation

The Heap size allocated for the application depends on the application performance requirements and available RAM memory.

Increasing the Heap size (with a higher number of packet buffers) boosts the application performance however this costs RAM memory size. Decreasing the Heap size saves more RAM memory space but the application performance is limited.

The memory allocation listed below is provided as example and it should be tailored to meet your application requirements. To ensure the robustness of the final application and to guarantee proper functioning in the worst case, you have to make sure that the application is tested in a network environment similar to the one that the device is to be linked to.

The heap size is configurable:

- For RVMDK projects: this variable is defined in the startup_stm32f10x_cl.s file as follows:

```
Heap_Size        EQU      0x00006400
```

- For EWARM projects: this variable is defined in the stm32f10x_flash.icf file as follows:

```
define symbol __ICFEDIT_size_heap__    = 0x6400
```

The Heap size can be retrieved at the application level using getHeapSize() function, that can be used for example to initialize the Heap when using memory management routines (such calloc1() and mem_free() which are the embedded systems version of standard calloc() & free() functions)

```
Heap_Size = getHeapSize();
```

STM32F107xx Ethernet low level driver optimization

In this same context, the STM32F107xx Ethernet low level driver (stm32_eth.c) provided within NicheLite has been optimized to remove the static buffers typically used as intermediate storage for the packets passed between the stack and the MAC. This zero copy implementation has the benefit of freeing more RAM memory and making it available to the application.

6 Debug tips

6.1 NicheTool diagnostic console

NicheTool is a command line driven interface (console) used as diagnostic suite for debugging, monitoring, and runtime configuration of the TCP/IP stack and applications.

The console uses a host PC serial utility (like HyperTerminal) and communicates with the target system through a UART based peripheral.

For STM32F107xx the following hardware and software resources are used to interface with the console:

- The USART2 interface is used in polling mode and is configured as follows: 115200 baud, 8-bits, 1 stop bit, no parity and no flow control.
- The getch() function used for any data read on the serial interface, declared in the in_stubs.c file, it uses the USART_ReceiveData() function of the stm32f10x_usart.c driver.
- The dputchar() function used to send data through the serial interface, declared in the in_stubs.c file, it uses the USART_SendData() function of the stm32f10x_usart.c driver.

When the stack is running, NicheTool is initialized and the INET> prompt is displayed on the console. The line commands can be displayed as Help menu lists or simply entered directly at the prompt.

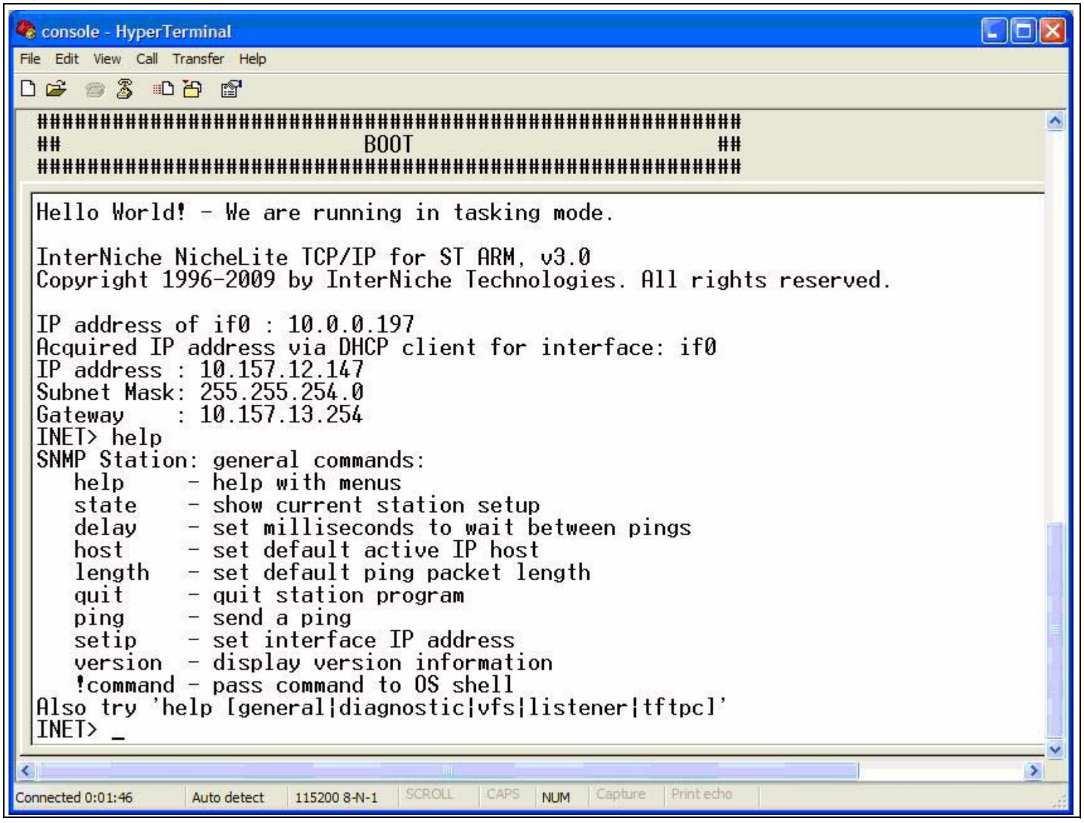
The debug commands are available when the option IN_MENUS is defined in the ipport.h file.

The most important debug commands are listed below:

- state - displays current setup information, interfaces, and default settings.
- tkstats - tasking system status
- buffers - display allocated packet buffer statistics
- queues - dump packet buffer queues
- linkstats - display link layer specific statistics
- arps - display ARP statistics and table
- ipstat - display IP layer statistics
- tcp - display TCP layer statistics
- udp - display UDP layer statistics
- icmpstat - display ICMP layer statistics

The *Figure 23* gives an example of the console running on the STM32F107xx.

Figure 23. NicheTool console



```
##############################################################
##                         BOOT                             ##
##############################################################
Hello World! - We are running in tasking mode.

InterNiche NicheLite TCP/IP for ST ARM, v3.0
Copyright 1996-2009 by InterNiche Technologies. All rights reserved.

IP address of if0 : 10.0.0.197
Acquired IP address via DHCP client for interface: if0
IP address : 10.157.12.147
Subnet Mask: 255.255.254.0
Gateway    : 10.157.13.254
INET> help
SNMP Station: general commands:
   help     - help with menus
   state    - show current station setup
   delay    - set milliseconds to wait between pings
   host     - set default active IP host
   length   - set default ping packet length
   quit     - quit station program
   ping     - send a ping
   setip    - set interface IP address
   version  - display version information
   !command - pass command to OS shell
Also try 'help [general|diagnostic|vfs|listener|tftpc]'
INET> _
```

Some commands are available depending on the stack configuration in the ipport.h file (for example, the ipstat command in supported only if NET_STATS is defined). For the complete list and description of available commands please refer to the NicheLite documentation.

For more information on how to implement your own system menu, please refer to *Section 7.3*.

6.2 Network statistics

The packets input/output statistics are accessed via the IP Management Information Base structure ip_mib, defined in the m_ip.c file, and provide useful information such as:

- Total number of datagrams received (including those received in error)
- Total number of datagrams transmitted (including those received in error)
- Number of input/output datagrams which were discarded (e.g., for lack of buffer space).
- …

For the complete description of the members of the ip_mib structure, please refer to the IpMib structure declaration in the ip.h file.

You can monitor the content of the ip_mib structure by placing it into the watch window of your debugger or by typing the ipstat command from the NicheTool console. Below is an example display from the NicheTool console:

```
INET> ipstat

IP MIB statistics:
```

```
Gateway: YES     default TTL: 64
rcv:  total: 35    header err: 0    address err: 18
rcv:  unknown Protocls: 0    delivered: 17
send: total: 3    discarded: 0     No routes: 0
```

6.3 Packet debugging

NicheLite provide the possibility to debug the packet buffer, this information is accessed via the pktlog structure defined in the pktalloc.c file. To enable this debugging, the NPDEBUG define should be set to 1 in ipport.h file.

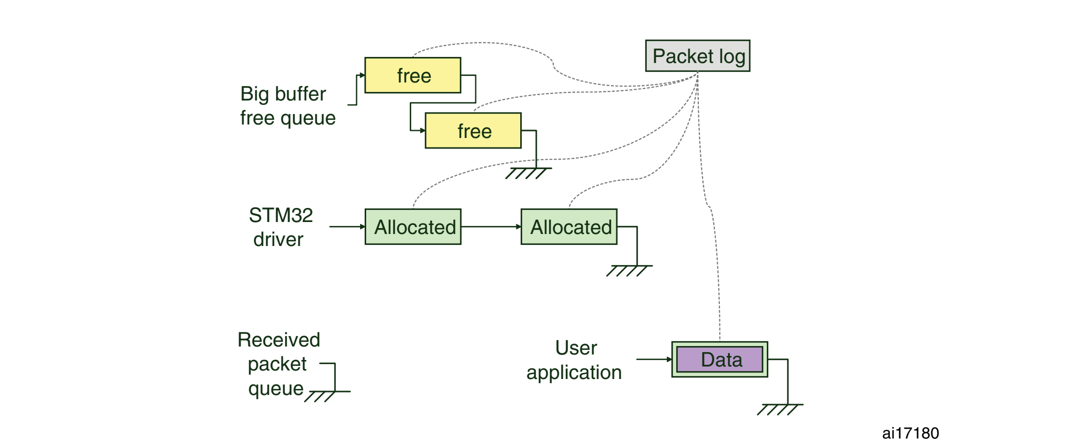
Note: The pktlog structure is used only for debug purposes, to handle packets reception and transmission the user application should use the dedicated API.

Once NPDEBUG is defined you can place the pktlog structure in the watch window of your debugger. Here is a list of some of the information you can get:

- Pointer to packet data, defined by the nb_buff member
- Length of the packet data, defined by the nb_blen member
- If the packet has data, the inuse member will be different from zero so that the packet is not moved into the free queue.
- IP address associated with packet, defined by fhost member
- TCP data and length are defined in m_data and m_len respectively
- …

For the complete description of pktlog structure's members, please refer to the netbuf structure declaration in the netbuf.h file.

Figure 24. Packet log



Another way to debug a packet buffer is to type the buffers command from the console, in this case you get more information about the buffer type and dump of its content…

```
INET> buffers
PACKET    len  buffer    que data data_offset is:0
20001390,1527,200013D4,big:40 40 01 00 5E 00 00 02 00 13 60 74 @@..^.....`t
200019D0,1527,20001A14,big:63 69 01 80 C2 00 00 00 00 14 1C DF ci..........
```

```
20002010,1527,20002054,non:FF 41 00 53 54 91 00 02 00 13 C4 A8 .A.ST.......
20002650,1527,20002694,big:FF 41 00 53 54 91 00 02 00 13 C4 A8 .A.ST.......
20002C90,1527,20002CD4,big:63 69 00 00 0C 07 AC 0C 00 53 54 91 ci.......ST.
200032D0,1527,20003314,big:40 40 00 53 54 91 00 02 00 13 C4 A8 @@.ST.......
20003910,1527,20003954,big:FF 41 00 53 54 91 00 02 00 13 C4 A8 .A.ST.......
20003F50,1527,20003F94,big:FF 41 00 00 0C 07 AC 0C 00 53 54 91 .A.......ST.
20004590,1527,200045D4,big:40 40 01 00 5E 00 00 05 00 13 60 74 @@..^.....`t
20004BD0,1527,20004C14,big:40 40 00 53 54 91 00 02 00 13 C4 A8 @@.ST.......
20005210,1527,20005254,non:63 69 00 00 0C 07 AC 0C 00 53 54 91 ci.......ST.
20005850, 205,20005894,lil:00 00 00 00 0C 07 AC 0C 00 53 54 91 .........ST.
20005968, 205,200059AC,lil:00 00 00 00 0C 07 AC 0C 00 53 54 91 .........ST.
20005A80, 205,20005AC4,lil:00 00 00 00 0C 07 AC 0C 00 53 54 91 .........ST.
20005B98, 205,20005BDC,lil:00 00 00 00 0C 07 AC 0C 00 53 54 91 .........ST.
20005CB0, 205,20005CF4,lil:00 00 00 00 0C 07 AC 0C 00 53 54 91 .........ST.
20005DC8, 205,20005E0C,lil:00 00 00 00 0C 07 AC 0C 00 53 54 91 .........ST.
20005EE0, 205,20005F24,lil:00 00 00 00 0C 07 AC 0C 00 53 54 91 .........ST.
20005FF8, 205,2000603C,lil:00 00 00 00 0C 07 AC 0C 00 53 54 91 .........ST.
```

6.4 Socket and packet queues

The stack implements a set of queues for managing sockets, big and little buffers:

- msoq list, defined in the tcputil.c file, used to queue existing sockets: when a socket is created it is added to the tail of the queue and when it is deleted, the memory space is released from the queue.
- bigfreeq list, defined in the nrmenus.c file, used to queue big buffers: when a buffer is used by the stack it is moved out of this queue and back into it when it is no longer needed by the stack.
- lilfreeq list, defined in the nrmenus.c file, used to queue little buffers: when a buffer is used by the stack it is moved out of this queue and back into it when it is no longer needed by the stack.
- rcvdq list, defined in the m_ipnet.c file, used to queue all received packets.

You can put each list into the watch window of your debugger to access the following members:

- q_head: first element in queue
- q_tail: last element in queue
- q_len: number of actual elements in queue
- q_max: maximum number of elements that have been queued since the stack start
- q_min: minimum number of elements that have been queued since the stack start. When q_min is 0 it means that the stack has run out of packet buffers in the listed queue type at least once since the stack was initialized.

Additional information about other queues can be obtained by typing the queue command from the console:

```
INET> queue

bigfreeq: head:20003910, tail:200032D0, len:9, min:8, max:11

lilfreeq: head:20005968, tail:20005850, len:11, min:10, max:11

rcvdq: head:00000000, tail:00000000, len:0, min:0, max:0
```

The line about rcvdq indicates how big the packet receive queue has gotten. rcvdq len reports the number of packets in the receive queue that have not yet been processed by the IP layer. rcvdq max indicates how high len has gotten since the stack start. A high value in rcvdq max would be an indication that the stack is not processing the receive queue in a timely manner.

7 NicheTask OS

7.1 Overview

The NicheLite TCP/IP stack presents two operating modes. Both modes allow you to run multiple processes or tasks simultaneously.

7.1.1 Superloop (OS disabled)

In this mode, all the tasks share a common stack. Each task is a single function call. To execute a task you have to call its corresponding function. This mode is the most efficient in terms of memory usage:

- Use of only one stack
- Less memory space is wasted
- With the OS disabled there are no OS RAM requirements (TCBs and other structures) which use valuable RAM resources.

The disadvantage of this mode is that no tasks can be created or deleted at runtime, it is a static mode. In superloop mode, each task function must return to the main loop after finishing execution.

A superloop-based application will look like this:

```
main(void)
{
...
#ifdef SUPERLOOP
   {
      char *cp;

      cp = pre_task_setup(); /* Perform any Pre-task initialization */
      if (cp)
         panic(cp);

      e = prep_modules(); /* Prepare and intialize the modules */

      netmain_init_done = 1;

      netmain_init(); /* Start all modules */

      cp = post_task_setup();
      if (cp)
         panic(cp);
   }
#endif
}
```

7.1.2 Multitasking system (OS enabled)

This mode uses the NicheTask multitasking scheduler software. This multitasking system contains only task control logic - no semaphores, mailboxes, etc. Tasks are scheduled in a round-robin manner and are not preempted. Once it gains control, each task runs until it voluntarily blocks. It is up to you to ensure that tasks do not run for indefinite amounts of time by relinquishing the CPU via calls to tk_block() or tk_next(), or one of the blocking TK_ macros.

Tasks can be created and deleted dynamically by calls to the tasking API. Each task has a stack and a task control structure. Tasks are created via the routine tk_new(), which returns a pointer to this structure. This pointer is thereafter used as a task ID.

On task creation, a new stack is allocated from heap (RAM memory) for that task. The size of each stack is static and determined at compile time. When a task is killed, the allocated stack memory is released.

There is a task control structure associated with each task instance, and a defined type task (defined in task.h file), which is a synonym of that structure. Each task structure contains a pointer to the tasks stack space.

```
struct task

{

   struct task *  tk_next;     /* pointer to next task */

   stack_t *   tk_fp;          /* task's current frame ptr */

   char *      tk_name;        /* the task's name */

   int         tk_flags;       /* flag set if task is scheduled */

   unsigned long      tk_count;       /* number of wakeups */

   stack_t *   tk_guard;       /* pointer to lowest guardword */

   unsigned    tk_size;        /* stack size */

   stack_t *   tk_stack;       /* base of task's stack */

   void *      tk_event;       /* event to wake blocked task */

  unsigned long     tk_waketick;   /* tick to wake up sleeping task
*/

};
```

The task control structure forms a circular list, chained by the tk_next member of the structure. The scheduler is a round robin scheduler; it simply loops through the circular list of tasks until it finds one which is ready to run (that is, a task whose tk_flags member value is equal to TF_AWAKE) and then switches to that task. A context switch merely consists of saving a small amount of state on the stack and calling the routine tk_switch() (which is called by tk_block()).

When a task stack is allocated, it is filled with guardword: a predefined constant used to track stack usage. The tk_guard field points to the last (lowest on STK_TOPDOWN systems) word on the stack. On every context switch, this word is checked to verify that it still contains a guardword. If it doesn't, the tasking scheduler assumes that the task had a stack overflow and aborts the system with a call to the panic() routine. Guardwords are also used by the tk_stats() function to determine how much of the stack has been used (you can invoke the kstats command from NicheTool console to get this information).

There is a global variable, tk_cur, which is a pointer to the task control structure of the task which is currently running.

For complete documentation about the NicheTask OS you can refer to
http://www.nichetask.com/

7.2 Configuring the NicheTask OS

The NicheLite stack operating mode is selected in the ipport.h file, two defines are available to choose between superloop and multitasking modes:

Superloop (OS disabled)

In ipport.h file, define SUPERLOOP and uncomment INICHE_TASKS

```
//#define INICHE_TASKS    1  /* InterNiche multitasking system */

#define SUPERLOOP       1  // InterNiche Superloop system
```

Multitasking system (OS enabled).

In ipport.h file, define INICHE_TASKS and uncomment SUPERLOOP

```
#define INICHE_TASKS    1  /* InterNiche multitasking system */

//#define SUPERLOOP       1  // InterNiche Superloop system
```

When NicheTask is enabled, you have to define the stack sizes for each task, in the osport.h file. The stack size must take into account, in addition to the requirements of the corresponding task, the need to have additional memory for any eventual interrupt. Since any task can be running when an interrupt occurs, the context for the interrupt is stored in the current running task's stack. The stack size must be larger than the task requirements.

A global variable cticks, defined in the ipport.h file, is used by the OS and incremented periodically for sleep timing controlled by the TK_SLEEP macro. The ctciks variable is incremented each time the SysTick ISR is executed (SysTick_Handler). The SysTick timer is configured in the clock_init() function, declared in the timer.c file, to fire at regular intervals and is used for time base generation.

7.3 Implementing your own system menus and tasks

You can insert menu items using the menu system API. This API is contained in the menus.c and nrmenus.c files for initial menu items, and all the associated functions for each menu item are contained in the menulib.c and nrmenus.c files.

The menu is defined as structure of type menu_op, defined in the menu.h file, and includes menu name, a pointer to a function that is called to execute the menu and a short description string. All menus are regrouped in a master list.

Then install_menu() is called by application code to add a new menu to the master list, the passed parameter is a pointer to a menu item.

The following is an example based on the Client/Server application demo

7.3.1 Task declaration (in demo_server.c file)

These macros declare a to_serverport pointer to a task structure then declare a task function called tk_serverport

```
TK_OBJECT(to_serverport);

TK_ENTRY(tk_serverport);
```

Task structure parameters declaration

```
struct inet_taskinfo servertask = {
 &to_serverport,      /* Returns pointer to the task control block */
 "Application server", /* Task name */
 tk_serverport,     /* Pointer to task function    */
 NET_PRIORITY,     /* Task priority  */
 WEB_STACK_SIZE,   /* task size */
};
```

7.3.2 Task function definition (in demo_server.c file)

Entry point of the task function created

```
  TK_ENTRY(tk_serverport)    /* entry point */
{
```

Wait for the main task to be started while (!iniche_net_ready)

```
      TK_SLEEP(1);
```

Task code

```
for (;;)
   {
........
      tk_yield();        /* give up CPU in case it didn't block */
      if (net_system_exit)
      {
         if (server_state == SERVER_RUNNING)
         {
            m_close(server_sock);
            server_state = SERVER_STOPPED;
         }
         break;
      }
   }
```

This macro is used in place of a return statement in the task routines declared with TK_ENTRY()

```
   TK_RETURN_OK();
}
```

7.3.3 Task creation (in tk_misc.c file)

This translates the inet_taskinfo argument “servertask” into a call to tk_new(). A pointer to the newly created task is returned by tk_ptr pointer.

```
e = TK_NEWTASK(&servertask); /*add servertask descriptor to the task
control block */
```

Error handler to the task creation:

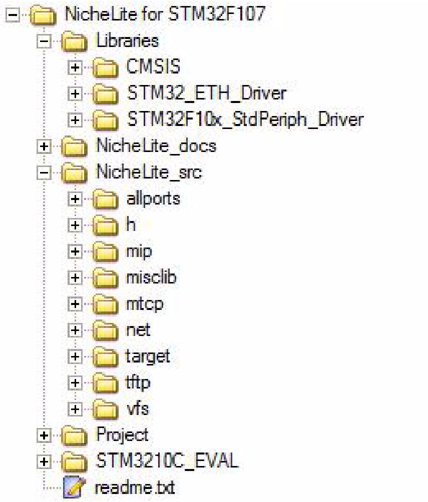
```
if (e != 0)
    {
      dprintf("servertask create error\n");
      panic("create_apptasks");
      return -1;  /* compiler warnings */
    }
```

8 NicheLite package for STM32F107xx

8.1 File structure

The NicheLite package for STM32F107xx connectivity line microcontrollers is supplied in a single zip file. The extraction of the zip file generates a folder, NicheLite for STM32F107xx, which contains the subfolders shown in *Figure 25* and described below.

Figure 25. NicheLite package file structure



```
NicheLite for STM32F107
    Libraries
        CMSIS
        STM32_ETH_Driver
        STM32F10x_StdPeriph_Driver
    NicheLite_docs
    NicheLite_src
        allports
        h
        mip
        misclib
        mtcp
        net
        target
        tftp
        vfs
    Project
    STM3210C_EVAL
    readme.txt
```

- **Libraries folder**
  - STM32_ETH_Driver subfolder: contains the C source for STM32F107xx Ethernet low level driver (stm32_eth.c/.h files).
  - CMSIS subfolder: contains the STM32F10x CMSIS files
  - STM32F10x_StdPeriph_Driver subfolder: contains C sources for STM32F10x Standard Peripherals drivers
- **NicheLite_docs folder:** contains NicheLite html documentation.
- **NicheLite_src folder**: contains NicheLite code source files.
  - allports and target subfolders[a]: contains all of the port-specific code necessary to run NicheLite, with the NicheTask OS on STM32F107xx
  - h subfolder[b]: contains NicheLite header files
  - mip subfolder[c]: contains the files implementing the IP, ARP, ICMP and most part of the UDP protocols
  - misclib subfolder[c]: several miscellaneous files
  - mtcp subfolder[c]: contains the files implementing the mini socket and TCP protocols
  - net subfolder[c]: contains file implementing the DHCP, DNS protocols and the buffer management
  - tftp subfolder[c]: contains the files implementing the TFTP protocol
  - vfs subfolder[c]: implementation of the virtual file system used by the NicheLite stack
- **STM3210C_EVAL folder**: contains LCD driver for STM3210C_EVAL board

- **Project folder**[a]: contains the Demos source files and preconfigured projects for EWARMv5 (IAR) and RVMDK (Keil) toolchains.

8.2 Description of available demonstration programs

The NicheLite package for STM32F107xx connectivity line microcontrollers comes with a rich set of TCP/IP stack protocol implementations:

- **DHCP Client**: once the STM3210C-EVAL board is connected to a DHCP server it gets its IP address and displays it on the LCD
- **Ping App**: the Ping application is available to check that the STM3210C-EVAL board is responding to ping requests.
- **HTTP**: this module is used to display an html page once the STM3210C-EVAL board IP address (given by the DHCP) is typed in the web browser. This html page includes a presentation of the embedded protocols in the NicheLite stack (used in this Demo) and a status bar displaying instantaneously the potentiometer (RV1) analog input conversion (available on STM3210C-EVAL board). The user can modify the potentiometer position and the status bar shape is updated accordingly.
- **TFTP client**: this part of the Demo, lets the user PUT and GET files from the STM3210C-EVAL board to the remote PC
- **TFTP server**: this part of the Demo lets the user PUT and GET files from the remote PC to the STM3210C-EVAL board.
- **UDP/TCP Client/Server**: this is a simple example based on a minimum of two STM3210C-EVAL Boards. One board is Server and all others are Clients. It provides source code examples for UDP and TCP connections.

All Demo applications are managed with the embedded NicheTask OS and are based on embedded sub-layer protocols (ARP, ICMP, UDP, and TCP).

For the complete list of the hardware configuration needed to run these demos, please refer to the “Hardware environment” section of the readme.txt file provided within this package.

Section 8.3 describes in detail the implementation of the UDP/TCP Client/Server. You can use this example as a starting point and then tailor it to your application requirements.

[a] These folders contains target dependant files; files that are modified to port the stack to STM32F107xx

[b] This directory contains both target independent and dependent files. ipport.h file is the most widely included target file, please refer to *Section 4.1* for more details.

[c] These folders contain target independent files; files that are mostly identical from one port to another

8.3 UDP/TCP client/server demo

This demo implements a simple client/server private communication protocol (data exchange, 2 bytes data length):

- One server board and one or more client boards are required to run this example. Both can be directly connected with a crossover cable or plugged to a network with straight cables.
- When connected on a public network, both boards get dynamic IP addresses from a DHCP server.
- On the client board, when the Key button (B3) is pushed, the application sends a TCP frame with 2 bytes of data on an opened connection. Depending on the byte values, it requests the Server application to set or reset LED4. The client application sets or resets its LED4 only after the answer from the server.

This example can be modified in order to send up to Max Segment Size of data to the server application.

For this example, both client and server PORT numbers have been fixed to 0xD903 (55555).

Prerequisites:

1. Make sure that client and server boards have different MAC addresses. For each board you have to modify the value of the MAC_ADDRx (x = 1 to 5) defines in the stm32.c file for the MAC address setting.
2. The board is configured as Server or Client depending on PE13 pin level after reset:
   a) PE13 pin level ow (connected to GND): board is configured as Client.
   b) PE13 pin level high (connected to VDD): board is configured as Server.

8.3.1 Client processing

When the client application starts, it doesn't know the server IP address, so the TCP connection with the server can't directly be opened.

To get the server IP address, Client opens an UDP connection on defined and known port number.

```
udp_open(0L, CLIENT_PORT,
CLIENT_PORT,udp_client_upcall,((void*)0xFFFFFFFD));
```

Where;

- 1st parameter is the destination IP address. 0L means all.
- 2nd parameter is the destination port number. The Server must listen to it.
- 3rd parameter is the local open port number.
- 4th parameter is the callback. This function is called when the UDP frame arrives on the defined port number.
- The last parameter is for stack internal use.

It returns a pointer to the UDP communication structure or 0 if error.

Then the client sends a server address message request (Data=99).

```
udp_send(CLIENT_PORT, CLIENT_PORT, pkt);
```

```
And the destination IP address set to broadcast message
```

```
pkt->fhost = 0xFFFFFFFF
```

Task state is set to CONNECTING state; it is modified to CONNECTED state from UDP callback when the application receives the UDP Server answer.

Note that Client has to start after Server to be sure server get UDP request.

For this example of a “private” protocol implementation, only data=98 received is processed. It is the server answer sent in response to the data=99 request in UDP_ServerAddrReq. It is used to get the server address to be able to communicate through TCP:

```
if(pkt->nb_prot[0] == '9'&& pkt->nb_prot[1] == '8')

{ /* Server addr answer*/

    applicationServerAddr = pkt->fhost;  /* extract server address */

    ...

}
```

Socket parameters

The application creates a socket for TCP communication on a defined port number and server IP address.

```
client_sin.sin_family = AF_INET;

client_sin.sin_addr.s_addr = applicationServerAddr; /*Server IP
address*/

client_sin.sin_port = CLIENT_PORT;/*port used by the application
example*/
```

Socket creation

```
client_sock = m_socket();
```

Connection parameters

```
client_sock->so_options |= SO_KEEPALIVE;

client_sock->lport = CLIENT_PORT;

client_sock->state |= SS_NBIO; /* Non blocking socket !!!!*/
```

Keepalive is enabled to check that the link between client and server is operating.

The same port number is used for client (local) and server (remote) ports.

It is mandatory to set the connection state as non blocking to be able to process the server connection acknowledge from the server.

Open connection

This sends a connection request to the server.

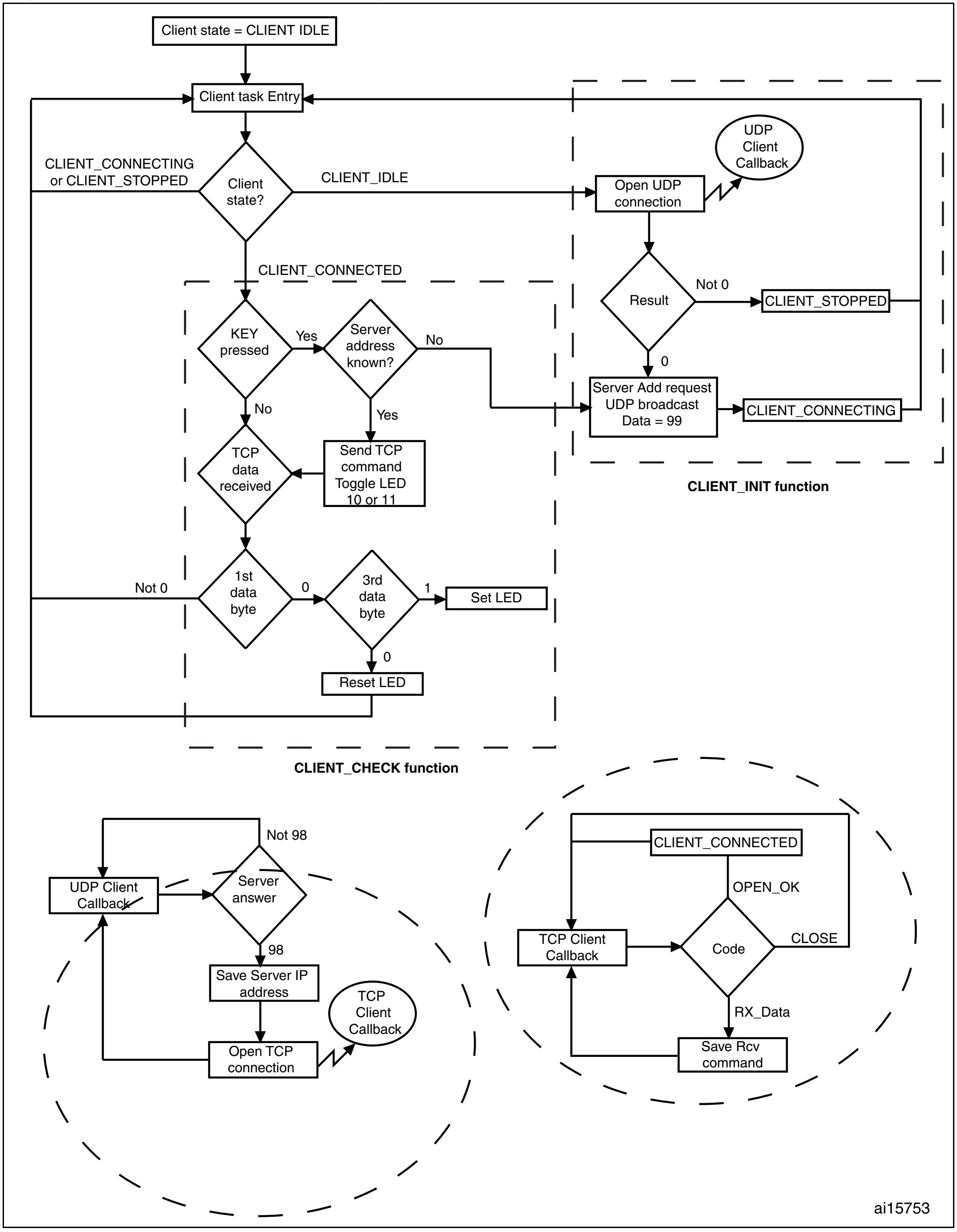
```
m_connect(client_sock, &client_sin, client_upcall);
```

TCP callback function

client_upcall is the TCP client callback, it processes all TCP frames received on CLIENT_PORT from the server.

The client processing is handled in the demo_client.c file and is presented in *Figure 26 on page 49*.

Figure 26. Client task processing



The client task loops and calls the client_init() or client_check() functions depending on the value of the Client_state variable.

The udp_client_upcall() and client_upcall() functions asynchronously process UDP and TCP received frames.

8.3.2 Server processing

The server process is easier than the client one.

The server task loops and calls the server_init() or server_check() functions depending on the value of the Server_state variable.

The task entry point is Server State = SERVER_IDLE.

The server_init function is called first.

The Server_init() function opens a UDP connection for the Client application requests to get the server IP address.

```
udp_open(0L, SERVER_PORT, SERVER_PORT,udp_server_upcall,
((void*)0xFFFFFFFD));
```

Where;

- 1st parameter is the destination IP address. 0L means all.
- 2nd parameter is the destination port number. Client must send request on it.
- 3rd parameter is the local open port number.
- 4th parameter is the callback. This function is called when UDP frame arrives on defined port number.
- The last parameter is for stack internal use.

It returns a pointer to the UDP communication structure or 0 if error.

The server_init function also opens the TCP connection.

Socket parameters

The application creates a socket for TCP communication on the defined port number and for every IP address.

```
   server_sin.sin_family=AF_INET;

   server_sin.sin_addr.s_addr = INADDR_ANY;     /* wildcard IP
address */

   server_sin.sin_port = SERVER_PORT;    /* port used by the
application example*/
```

Open connection

The server application opens a socket and waits for a connection request from client.

```
   m_listen(&server_sin, server_upcall, &e);
```

Callback functions

The udp_server_upcall and server_upcall functions asynchronously process UDP and TCP received frames.

The udp_server_upcall function manages UDP request at packet level.

```
    udp_server_upcall(PACKET pkt, void * data)
```

For example, to receive server IP address, the client application sends a UDP request on SERVER_PORT with data = “99”. The server application answers with data = “98”.

```
  udp_send(SERVER_PORT, SERVER_PORT, pkt_send)
```

The server_upcall callback function manages the TCP request at socket level.

```
    server_upcall(int code, M_SOCK so, void * datapkt)
```

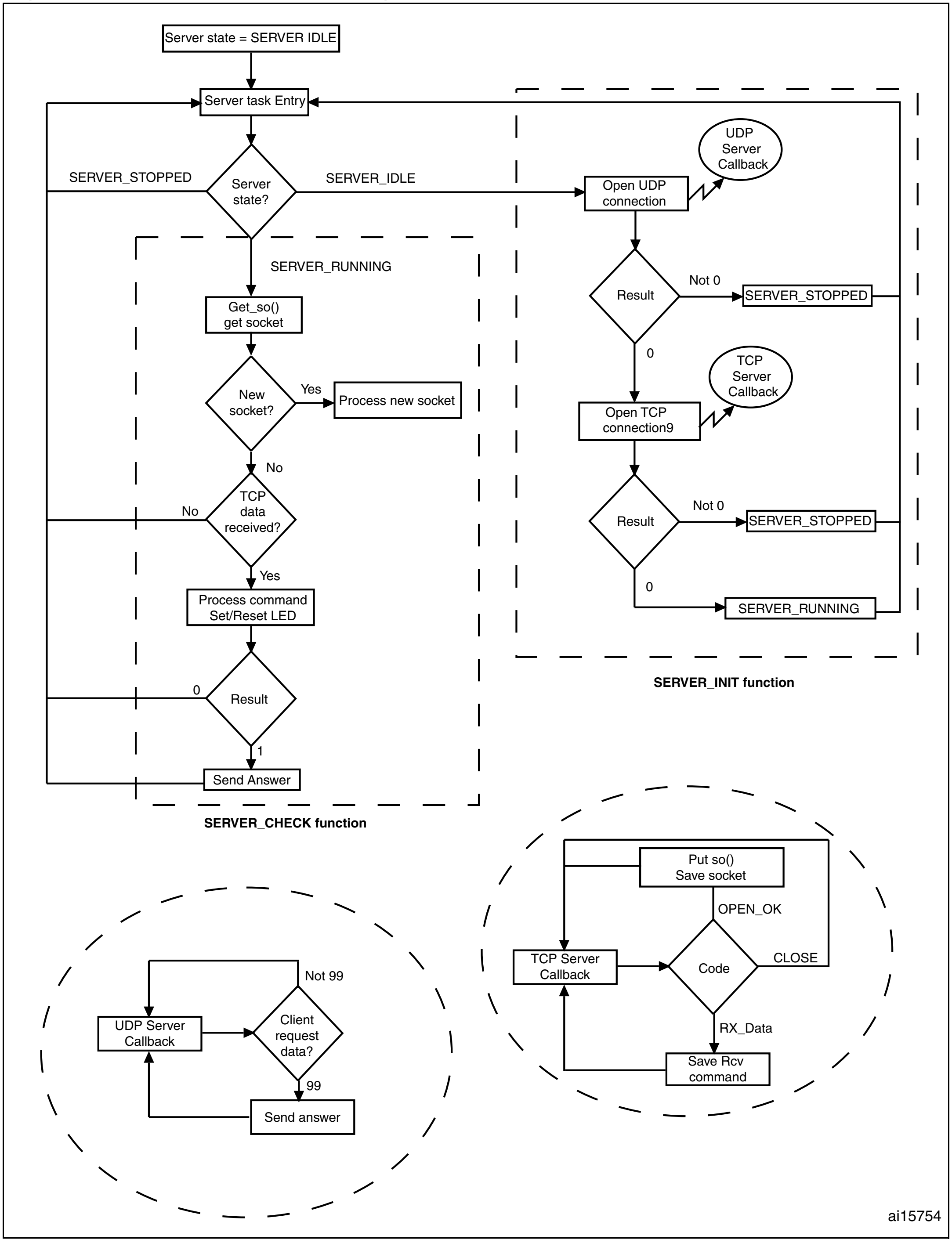
Where:

- 1st parameter code is the status of the connection
- 2nd parameter is the socket structure
- The last parameter can be a connection request from the client or data received on the socket from the client.

The server_upcall callback doesn't directly process this information; it saves information that is processed during the server_check function call.

The server processing is handled in the demo_server.c file and is presented in *Figure 27 on page 52*.

Figure 27. Server task processing



9 Revision history

Table 4. Document revision history

Date	Revision	Changes
06-Aug-2009	1	Initial release.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan -
Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com