µC/TCP-IP™

The Embedded Protocol Stack

User's Manual

Micriµm Press

Weston, FL 33326

µC/TCP-IP User's Manual

1. UserManual . . . 3
  1.1 About . . . 6
  1.2 Getting Started . . . 11
    1.2.1 Installing . . . 12
    1.2.2 Building and Running the Sample Application . . . 15
      1.2.2.1 Wired Ethernet Interface Setup . . . 19
      1.2.2.2 WiFi Interface Setup . . . 24
  1.3 Directories and Files . . . 29
    1.3.1 TCPIP Network Devices . . . 38
    1.3.2 TCPIP Network Interface . . . 41
    1.3.3 TCPIP Network File System Abstraction Layer . . . 43
    1.3.4 TCPIP Network CPU Specific Code . . . 45
    1.3.5 TCPIP Network CPU Independent Source Code . . . 47
    1.3.6 TCPIP Network Security Manager CPU Independent Source Code . . . 48
    1.3.7 TCPIP Network Examples Code . . . 49
  1.4 Configuration . . . 51
    1.4.1 Network Stack Configuration . . . 52
    1.4.2 Network Tasks Configuration . . . 62
    1.4.3 Network Interface Configuration . . . 65
    1.4.4 LIB Memory Heap Configuration . . . 81
  1.5 TCPIP Initialization Guide . . . 85
    1.5.1 Prerequisite module initialization . . . 86
    1.5.2 Initializing Tasks and objects . . . 88
    1.5.3 Initializing Interfaces . . . 89
    1.5.4 IP Address Configuration . . . 96
    1.5.5 Initializing+Shell+commands . . . 103
    1.5.6 Sample applications . . . 105
      1.5.6.1 Ethernet Sample Application . . . 106
      1.5.6.2 WiFi Sample Application . . . 113
      1.5.6.3 Multiple Interfaces Sample Application . . . 124
  1.6 Network Board Support Package . . . 129
    1.6.1 Ethernet BSP Layer . . . 130
    1.6.2 Wireless BSP Layer . . . 136
    1.6.3 Specifying the Interface Number of the Device ISR . . . 145
  1.7 Troubleshooting Guide . . . 146
    1.7.1 Initialization Issues . . . 147
    1.7.2 Application Issues . . . 153
    1.7.3 Performance Issues . . . 155
    1.7.4 Statistics and Error Counters . . . 159
    1.7.5 Architecture . . . 162
      1.7.5.1 Module Relationships . . . 164
      1.7.5.2 Network Buffer Architecture . . . 169

1.7.5.3 TCPIP Tasks and Priorities 171
1.7.5.4 Receiving a Packet 173
1.7.5.5 Transmitting a Packet 176
1.7.5.6 Timer Management 178

UserManual

Version 3.03.00

µC/TCP-IP is a compact, reliable, high-performance TCP/IP protocol stack. Built from the ground up with Micrium’s unique combination of quality, scalability and reliability, µC/TCP-IP, the result of many man-years of development, enables the rapid configuration of required network options to minimize time to market.

The source code for µC/TCP-IP contains over 100,000 lines of the cleanest, most consistent ANSI C source code available in a TCP/IP stack implementation. C was chosen since C is the predominant language in the embedded industry.

Portable

C/TCP-IP is ideal for resource-constrained embedded applications. The code was designed for use with nearly any CPU, RTOS, and network device. Although C/TCP-IP can work on some 8 and 16-bit processors, C/TCP-IP is optimized for use with 32 or 64-bit CPUs.

Scalable

The memory footprint of C/TCP-IP can be adjusted at compile time depending on the features required, and the desired level of run-time argument checking appropriate for the design at hand. Since C/TCP-IP is rich in its ability to provide statistics computation, unnecessary statistics computation can be disabled to further reduce the footprint.

Coding Standards

Coding standards were established early in the design of C/TCP-IP. They include:

- C coding style
- Naming convention for #define constants, macros, variables and functions
- Commenting

- Directory structure

These conventions make C/TCP-IP the preferred TCP/IP stack implementation in the industry, and result in the ability to attain third party certification more easily as outlined in the next section.

MISRA C

The source code for C/TCP-IP follows Motor Industry Software Reliability Association (MISRA) C Coding Standards. These standards were created by MISRA to improve the reliability and predictability of C programs in safety-critical automotive systems. Members of the MISRA consortium include such companies as Delco Electronics, Ford Motor Company, Jaguar Cars Ltd., Lotus Engineering, Lucas Electronics, Rolls-Royce, Rover Group Ltd., and universities dedicated to improving safety and reliability in automotive electronics. Full details of this standard can be obtained directly from the MISRA web site at: www.misra.org.uk.

Safety Critical Certification

C/TCP-IP was designed from the ground up to be certifiable for use in avionics, medical devices, and other safety-critical products. Validated Software’s Validation SuiteTM for C/TCP-IP will provide all of the documentation required to deliver C/TCP-IP as a pre-certifiable software component for avionics RTCA DO-178B and EUROCAE ED-12B, medical FDA 510(k), IEC 61508 industrial control systems, and EN-50128 rail transportation and nuclear systems. The Validation Suite, available through Validated Software, will be immediately certifiable for DO-178B Level A, Class III medical devices, and SIL3/SIL4 IEC-certified systems. For more information, check out the C/TCP-IP page on the Validated Software web site at: www.ValidatedSoftware.com.

If your product is not safety critical, however, the presence of certification should be viewed as proof that C/TCP-IP is very robust and highly reliable.

RTOS

C/TCP-IP assumes the presence of an RTOS, yet there are no assumptions as to which RTOS to use with C/TCP-IP. The only requirements are that it must:

- Be able to support multiple tasks

- Provide binary and counting semaphore management services
- Provide message queue services

Micrium provides an Kernel Abstraction layer that allows for the use of almost any commercial or open source RTOS. It can be found under the KAL folder of the µC/Common directory. Details regarding the RTOS are thus hidden from C/TCP-IP. KAL includes the encapsulation layer for C/OS-II and C/OS-III real-time kernels.

About

This chapter presents a quick introduction to the Micriµm Internet protocol suite also commonly known as µC/TCP-IP.

TCP/IP Layer Model

Layers

The Internet Protocol suite regroups all the protocols used for communication accross the Internet or similar networks.

A TCP/IP stack is divided into layers of same functionnality. Each communication protocol belongs to one of the layers. Table - TCP/IP Layer Model shows the differents layers of the TCP/IP model and also the OSI model. At the rigth, a list of the protocols associated with each layer that the Micriµm µC/TCP-IP stack supports.

OSI Model	TCP/IP Model (Dod)	µC/TCP-IP Internet Protocol Suite			
Application	Application	HTTP, FTP, DNS, SMTP, POP3, SNTP, TELNET, TFTP, SSL, DHCP			
Presentation					
Session					
Transport	Transport	TCP, UDP			
Network	Internet	IPv4, IGMP, ICMPv4, ARP		IPv6, MLDP, ICMPv6, NDP	
DataLink	Network Access	IF / 802x			
		Ethernet		WiFi	
Physical		Ethernet Driver	PHY Driver	WiFi Driver	WiFi Manager

Table - TCP/IP Layer Model

µC/TCP-IP Protocols

µC/TCP-IP consists of the following protocols:

- Device drivers

- Network interfaces (e.g., Ethernet, WiFi, etc.)
- Address Resolution Protocol (ARP)
- Neighbor Discovery Protocol (NDP)
- Internet Protocol (IPv4 and IPv6)
- Internet Control Message Protocol (ICMPv4 and ICMPv6)
- Internet Group Management Protocol (IGMP)
- Multicast Listener Discovery Protocol (MLDP)
- User Datagram Protocol (UDP)
- Transport Control Protocol (TCP)
- Sockets (Micrium and BSD v4)

Interfaces

Actually µC/TCP-IP supports Ethernet interface that use Ethernet frame and/or IEEE 802.4.

Also WiFi interfaces can be used as long as the device implements by itself IEEE 802.11. Basically, µC/TCP-IP only sends some commands to the module to scan for wireless networks, join or leave a specific network. The WiFi module must be able to encrypt and decrypt by itself all the network data. The network data between the host and the wireless module is transferred using IEEE 802.4. Currently only SPI is supported as communication bus between the host and the wireless module.

Devices

µC/TCP-IP may be configured with multiple-network devices and network (IP) addresses. Any device may be used as long as a driver with appropriate API and BSP software is provided. The API for a specific device (i.e., chip) is encapsulated in a couple of files and it is quite easy to adapt devices to µC/TCP-IP.

IPv4 & IPv6

IPv4

The Micriµm Network Stack supports IPv4 as described in RFC #791, with the following restrictions/constraints:

- ONLY supports a single default gateway per interface.
- IPv4 forwarding/routing NOT currently supported.
- Transmit fragmentation NOT currently supported.
- IPv4 Security options NOT supported.

The IPv4 Layer also implements:

- ICMPv4 protocol for Internet control messages;
- ARP protocol for link layer address resolution;
- IGMP protocol for multicast communication.

IPv6

The Micriµm Network Stack supports IPv6 as described in RFC #2460, with the following restrictions/constrains:

- IPv6 Extension Headers is NOT currently supported.

The IPv6 layer also implements:

- ICMPv6 protocol for Internet control messages;
- NDP protocol for link layer address resolution;

- MLDP protocol for mutlicast communication.

Socket API

The user application interfaces to µC/TCP-IP via a well known API called BSD sockets (or µC/TCP-IP's internal socket interface). The application can send and receive data to/from other hosts on the network via this interface. Many books and tutorials exist about BSD sockets programming, mostly the concepts explained in these reference can be applied to µC/TCP-IP socket programming.

Network Application Protocols

Micrium Add-ons

Micrium offers application layer protocols as add-ons to µC/TCP-IP. A list of these network services and applications includes:

µC/DCHPc	DHCP client
µC/DNSc	DNS client
µC/HTTPs	HTTP server (web server)
µC/FTPc	FTP client
µC/FTPs	FTP server
µC/SMTPc	SMTP client
µC/SNTPc	Network Time Protocol client
µC/TELNETs	Telnet server
µC/POP3c	POP3 client
µC/TFTPc	FTP client
µC/TFTPc	TFTP client
µC/IPerf	Network testing tool

BSD Based Application

Any well known application layer protocols following the BSD socket API standard can be used with µC/TCP-IP.

RFC validation

µC/TCPIP is regularly validated via a popular automated network validation library provided by Ixia and called IxANVL. It guaranteed that RFCs are always respected and correctly implemented.

Getting Started

This chapter gives you some insight into how to install and use the µC/TCP-IP stack. The following topics are explained in this chapter:

- Prerequisites
- Downloading the Source Code
- Installing the Files
- Building and Running the Sample Application

At the end of this chapter, you should be able to build and run your first TCP/IP application using the µC/TCP-IP stack.

Installing

Prerequisites

Before running your first application, you must ensure that you have the minimal set of required tools and components:

- Toolchain for your specific microcontroller.
- Development board.
- Source code for the μC/TCPI-IP stack.
- Source code for all the other Micriμm modules required by the μC/TCPI-IP stack (see section Additional Modules)
- Interface device driver compatible with your hardware for the μC/TCP-IP stack (Ethernet + PHY driver or WiFi driver).*
- Board support package (BSP) for your development board.*
- A running project for your selected RTOS.

> * If Micrium does not support your network device driver, you will have to write your own device driver. The same goes for your BSP. However, you can contact Micriμm (support@micrium.com) to see if a BSP example exists for your development board to help you get started. Refer to section μC/TCPIP Driver Manual for more information on writing your own Ethernet or WiFi device driver and section Network Board Support Package for writing your own BSP.

Additional Modules

μC/TCP-IP depends on other modules to run. First, it needs the presence of a RTOS. Furthermore, μC/CPU, μC/LIB and μC/Common modules are required.

If you are using one of the two Micriμm OS, complete documentation can be found here for μC/OS-II and here for μ/COS-III. Refer to those guides for more information on the requirements, installation, configuration and running of those RTOS.

The µC/CPU module regroups the processor's hardware-dependent code. µC/CPU includes many ports for all the different CPU architectures Micriµm supports. You must therefore used the port corresponding to your specific CPU. The complete µC/CPU documentation can be found here.

The µC/LIB module is the Micriµm run-time library, implementing the most common standard library functions, macros, and constants. The complete µC/LIB documentation can be found here.

The µC/Common repository comprises multiple modules required by the µC/TCP-IP stack. Among others, the KAL module is included. KAL stands for Kernel-Abstraction Layer. It is used by µC/TCP-IP stack and other Micriµm products to interacts with the RTOS by specifying a set of generic API functions. KAL comes with the µC/OS-II and µC/OS-III ports. The complete KAL documentation can be found here.

Downloading the source code

µC/TCP-IP stack can be downloaded from the Micrium customer portal as all the other required modules. The distribution package includes the full source code and documentation.

The customer portal also includes all the network interface device driver supported by Micriµm. Wired Ethernet devices and WiFi devices are supported by Micriµm. Download the device driver adequate for your project. If your device is not support by Micriµm, you will need to develop your own device driver. Refer to section µC/TCPIP Driver Manual for all the details.

You can log into the Micrium customer portal at the address below to begin your download (you must have a valid license to gain access to the file):

`http://micrium.com/login`

Installing the Files

Once all the distribution packages have been downloaded to your host machine, extract all the files at the root of your C:\ drive for instance. The package may be extracted to any location. After extracting all the files, the directory structure should look as illustrated in Figure - Directory Tree for μC/TCP-IP. In the example, all Micriμm products sub-folders shown in Figure - Directory Tree for μC/TCP-IP will be located in `C:\Micrium\Software\`. The Micriμm μC/OS-III RTOS has been chosen for this example.

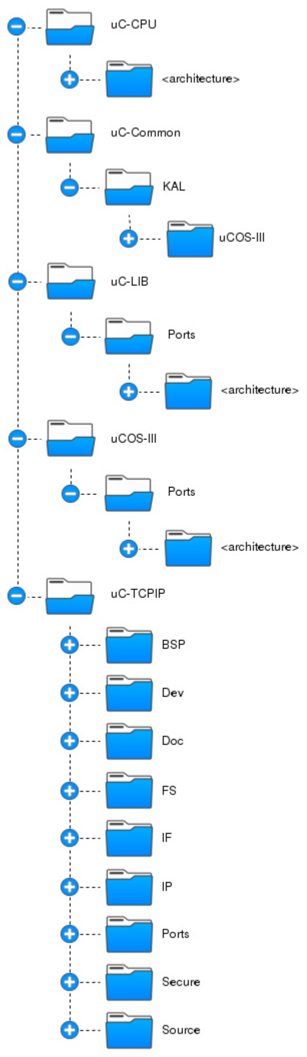

Figure - Directory Tree for μC/TCP-IP

Building and Running the Sample Application

This section describes all the steps required to build a TCPIP-based application. The instructions provided in this section are not intended for any particular toolchain, but instead are described in a generic way that can be adapted to any toolchain.

The best way to start building a TCPIP-based project is to start from an existing project. If you are using µC/OS-II or µC/OS-III, Micrium provides example projects for multiple development boards and compilers. If your target board is not listed on Micrium's web site, you can download an example project for a similar board or microcontroller.

Working project with RTOS

The first step before including the µC/TCP-IP stack is to have a working project with the RTOS of your choice. As previously mentioned, Micriµm offers starting example with the µC/OS-II and µC/OS-III kernels for many evalboards.

- Complete documentation for µC/OS-II, including a Getting Started Guide
- Complete documentation for µC/OS-III, including a Getting Started Guide

Including Additional Modules to the Project

Once you have a working project with your RTOS, additional modules are needed by the Micriµm TCP/IP stack that are not necessarily already included in your project. Therefore, be sure to add µC/CPU, µC/LIB and µC/Common to your project.

- Complete documentation for µC/CPU
- Complete documentation for µC/LIB
- Complete documentation for µC/Common

Bear in mind to include the required paths associated with those modules to your project's C compiler settings.

Including the Board Support Package (BSP)

In order for the network device driver to remain hardware independent, µC/TCPIP requires a BSP abstraction layer to implement such things as configuring clocks, interrupt controllers, general-purpose input/ouput (GPIO) pins. Inside your project folder tree, under the board name, you should already have a BSP folder. Boards for which TCPIP application have been develop should have a TCPIP folder inside the BSP directory. You can add this directory to your project tree. If your board does not have such a folder, you would have to write your own BSP for the network device driver. Refer to section Network Board Support Package for more details on how to write a BSP for µC/TCP-IP. Micrium offers template files inside the BSP folder in the µC/TCP-IP source code distribution to get you started with your own BSP. However, we recommend starting with a working configuration from an example project for your network device. Micrium might have some projects available only for internal usage, so if no working project are found online, please ask at support@micrium.com for a BSP file example specific for your network device.

Afterwords, add a path leading to the following include paths to your project's C compiler settings:

```
\Micrium\Software\EvalBoards\<manufactuer>\<boardname>\BSP\TCPIP
```

Including TCP-IP Stack Source Code

The µC/TCP-IP files to include in your project depends on the network interface(s) presents on your development board. Therefore, the complete files list we will be presented inside each following sub-sections associated with an interface type.

Ethernet Interface Setup

Wi-Fi Interface Setup

Copying and Modifying Template Files

Copy the files from the uC-TCPIP configuration folder into your application as illustrated in Figure - Copying Template Files .

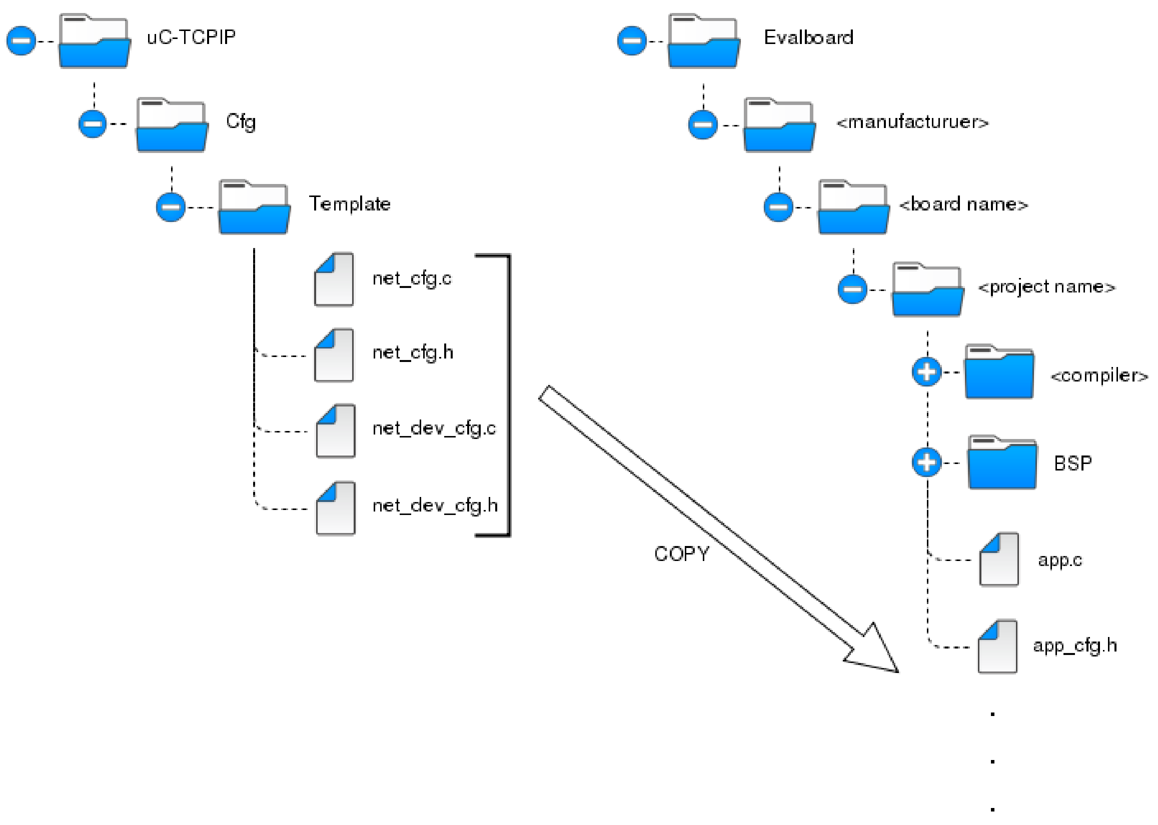

Figure - Copying Template Files

`net_cfg.c` is a configuration file including the `NET_TASK_CFG` objects used to configured the µC/TCP-IP internal tasks. The µC/TCP-IP stack has three internal tasks : the Receive task, the Transmit De-allocation task and the Timer task. Each task as its own `NET_TASK_CFG` object defining the task priority, the task's stack size and the pointer to start of task stack. Refer to section Network Tasks Configuration for more details on the µC/TCP-IP tasks configuration.

`net_cfg.h` is a configuration file used to setup µC/TCP-IP stack static parameters such as the number of available sockets, TCP connections and network timers, the ARP and NDP parameters, the number of configurable interface and so on. Refer to section Static Stack Configuration for more details on all the configurations inside `net_cfg.h`.

`net_dev_cfg.c` and `net_dev_cfg.h` are configuration files used to set the wired or wireless device interface parameters such as the number and size of network buffers available for transmission and reception and the base address of the device's registers. They also include the PHY parameters such as the PHY bus mode (RMII or MII) in the case of a wired Ethernet device.

Since the device configuration is different depending if your interface is wired or wireless, the details on the device configuration modifications will be shown in the corresponding sub-sections.

Modifying the Application Configuration Files

The µC/TCP-IP stack uses additional heap memory space. Therefore, it is possible that your example application will require more allocation of heap memory. If ti is the case, you can increase the value of the #define `LIB_MEM_CFG_HEAP_SIZE` inside the `lib_cfg.h` file of your example project. You can refer to section LIB Memory Heap Configuration for more details on the heap usage of the µC/TCP-IP stack.

Wired Ethernet Interface Setup

Including TCP-IP Stack Source Code

Include the following files in your project tree from the µC/TCP-IP source code distribution, as indicated in Figure - µC/TCP-IP Source Code. In this figure, the IP folder only show the sub-folder IPv4 as an example. If you are running with IPv6, please add the IPv6 folder instead or add both IPv4 and IPv6 folders if you want your project to support both IP version.

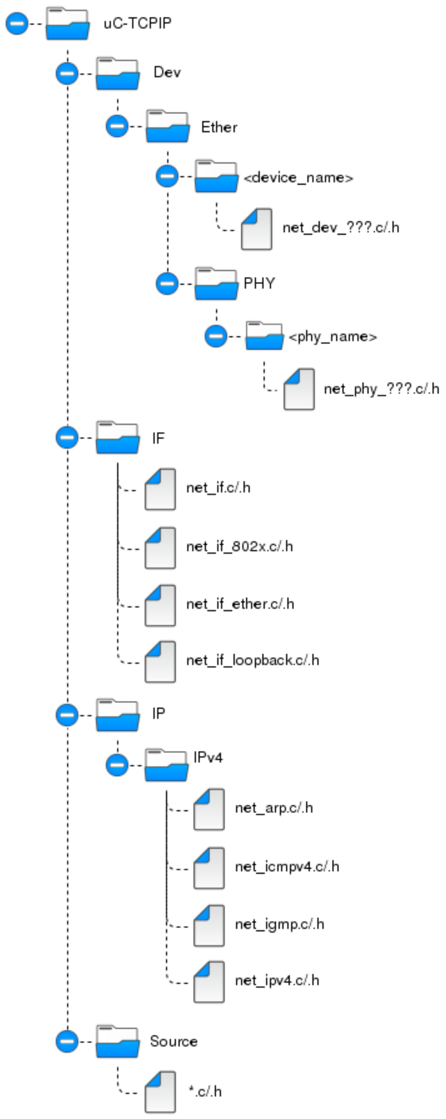

Figure - µC/TCP-IP Source Code

As indicated in the Figure - µC/TCP-IP Source Code, all the files in the Source folder must be added to your project tree. Furthermore, if a TCP-IP port exists for your CPU architecture inside the "Ports" folder, you can also include it to your project files.

Second, add the following include paths to your project’s C compiler settings:

```
\Micrium\Software\uC-TCPIP
\Micrium\Software\uC-TCPIP\Dev\Ether\<device_name>
\Micrium\Software\uC-TCPIP\Dev\Ether\PHY\<phy_device_name>
```

Modify the Interface Device Configuration

Modify Ethernet Device Configuration

Inside the net_dev_cfg.c file, there are different device configuration templates. Since that, in this section, the example project we want to run is with a Ethernet wired device, the configuration that interests us is beneath the section "EXAMPLE ETHERNET DEVICE CONFIGURATION".

Next you need to modify the Ethernet device configuration template as needed by your application. Refer to section Memory Configuration and Ethernet Interface Configuration for all the details on the parameters to configure.

Modify PHY Configuration

Under the Ethernet device configuration, you will also found the PHY device configuration. This configuration will also need to be adjusted according to your PHY setup on the development board. Refer to section Ethernet Interface Configuration for all the details on the PHY configuration.

Modify Static Configurations

As previously mentionned, the µC/TCP-IP static configurations are located in the `net_cfg.h` file. For this getting started guide, the template file without modification should be enough to get you started. Depending on your Interface device configuration, it is possible that you would need to adjust the µC/TCP-IP queues' configurations as listed in Listing - µC/TCP-IP Static Configuration Modifications. Refer to section Task Queue Configuration for more details on the TCP-IP queues's configurations.

```
#define  NET_CFG_IF_RX_Q_SIZE                       50
#define  NET_CFG_IF_TX_DEALLOC_Q_SIZE               50
```

Listing - µC/TCP-IP Static Configuration Modifications

Tasks Priority

The `net_cfg.c` file includes the three network task configurations. You will need to defined the priority of each of those tasks. The priorities will depend on the other tasks already present in your application. Refer to section Network Tasks Configuration for all the details on configuring the network tasks and their priority.

Example Project Setup

The purpose of this example project is to setup a network host on the target board to allow it to communicate with other hosts on the network. Figure - Example Application Setup shows the project test setup for a Ethernet wired interface. The target board is wire-connected to an Ethernet switch or via an Ethernet cross-over cable to a Windows-based PC. The PC's IP address is set to 10.10.10.111 and the target's addresses will be configure to 10.10.10.64 as it will be shown in the next section Adding µC/TCP-IP application function.

This example project contains enough code to be able to ping the board. Therefore, after successfully running the project, You will be able to issue the following command form a command-prompt:

```
ping 10.10.10.64
```

Ping (on the PC) should reply back with the ping time to the target. µC/TCP-IP target projects connected to the test PC on the same Ethernet switch or Ethernet cross-over cable achieve ping times of less than 2 milliseconds.

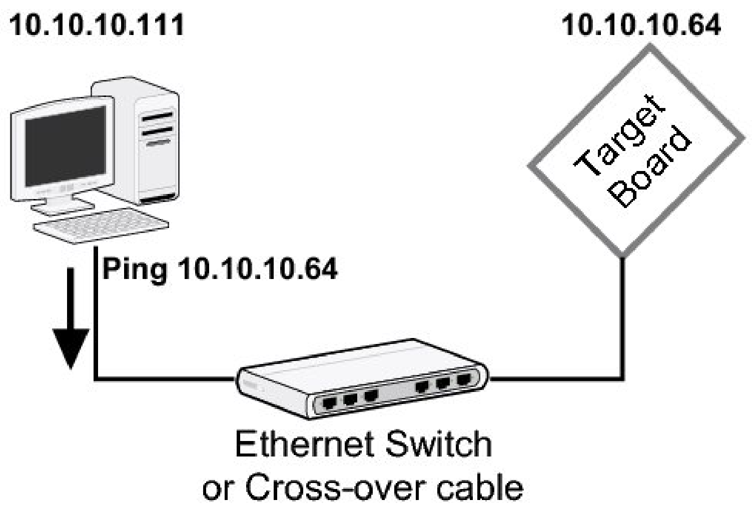

Figure - Sample Application Setup

After you have successfully completed and run the example project, you can use it as a starting point to run other µC/TCP-IP demos you may have purchased.

Adding Additional includes

Since the µC/TCP-IP module was added to the example project, the following include must be added to the app.c file :

```
#include  <KAL/kal.h>
#include <Source/net.h>
#include <net_dev_????.h>
#include <net_phy_????.h>
#include <net_dev_cfg.h>
#include <net_bsp.h>
```

Adding µC/TCP-IP Application Function

Before running the example application, you will need to add the new funtion, `AppInit_TCPIP()`, in your app.c file to initialize and setup the µC/TCP-IP stack. Section Tasks and Objects Initialization gives an example of the main application task inside which the `AppInit_TCPIP()` function will be called. Section Ethernet Sample Application gives an `AppInit_TCPIP()` example for a wired Ethernet interface.

Those code examples will need to be modified in accordance with your project setup. For example, when adding an interface, your network device configuration object name (inside `net_dev_cfg.c`) will need to be specify and the IP address used in the example could need to be change to match your network.

Once the source code is built and loaded into the target, the target will respond to ICMP Echo (ping) requests.

WiFi Interface Setup

Including TCP-IP Stack Source Code

Include the following files in your project tree from the µC/TCP-IP source code distribution, as indicated in Figure - µC/TCP-IP Source Code. In this figure, the IP folder only show the subfolder IPv4 as an example. If you are running with IPv6, please add the IPv6 folder instead or add both IPv4 and IPv6 folders if you want your project to support both IP version.

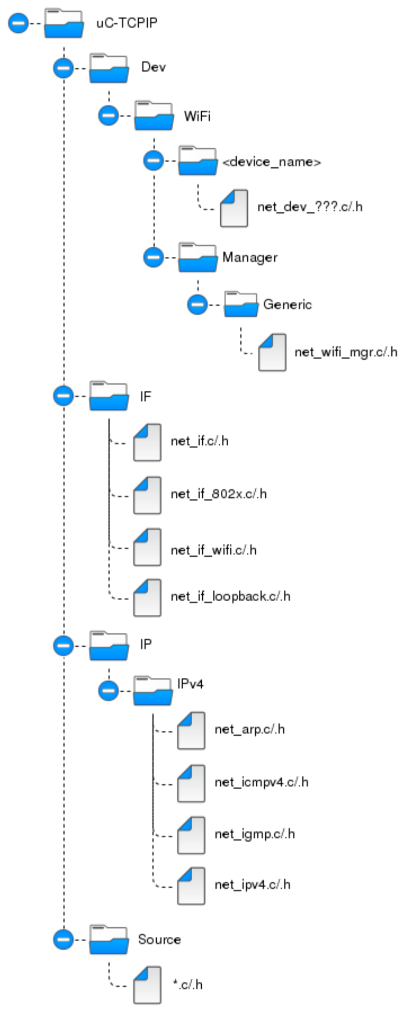

Figure - µC/TCP-IP Source Code

As indicated in the Figure - µC/TCP-IP Source Code, all the files in the Source folder must be added to your project tree. Furthermore, if a TCP/IP port exists for your CPU architecture inside de Ports folder, you can also include it to your project files.

Second, add the following include paths to your project’s C compiler settings:

```
\Micrium\Software\uC-TCPIP
\Micrium\Software\uC-TCPIP\Dev\WiFi\<device_name>
\Micrium\Software\uC-TCPIP\Dev\WiFi\Manager\Generic
```

Modify the Interface Device Configuration

Inside the `net_dev_cfg.c` file, there are different device configuration templates.

Since that, in this section, the example project we want to run is with a WiFi device, the configuration that interests us is beneath the section "EXAMPLE WIFI DEVICE CONFIGURATION".

Next, you need to modify the WiFi device configuration template as needed by your application. Refer to section Memory Configuration and Wireless Interface Configuration for all the details on the parameters to configure.

Modify Static Configurations

As previously mentionned, the µC/TCP-IP static configurations are located in the net_cfg.h file. For this getting started guide, the template file without modification should be enough to get you started. Depending on your Interface device configuration, it is possible that you would need to ajust the µC/TCP-IP queues' configurations as listed in Listing - µC/TCP-IP Static Configuration Modifications. Refer to section Static Stack Configuration for more details on the TCP-IP queues's configurations.

```
#define  NET_CFG_IF_RX_Q_SIZE                      50
#define  NET_CFG_IF_TX_DEALLOC_Q_SIZE              50
```

Listing - µC/TCP-IP Static Configuration Modifications

Tasks Priority

The net_cfg.c file includes the three network task configurations. You will need to defined the priority of each of those tasks. The priorities will depend on the other tasks already present in your applicaiton. Refer to section Network Tasks Configuration for all the details on configuring the network tasks and their priority.

Example Project Setup

The purpose of this example project is to setup a network host on the target board to allow it to communicate with other hosts on the network. Figure - Example Application Setup shows the project test setup for a target board with a WiFi interface. The target board has WiFi interface that allows the board to connect to a WiFi access point. In this example, a router acts as the access point and allows the PC to be on the same network as the board. The PC's IP address is set to 10.10.10.111 and the target's addresses wil be configure to 10.10.10.64 as it will be shown in the next section Running the Example Application.

This example project contains enough code to be able to ping the board. Therefore, after successfully running the project, You will be able to issue the following command form a command-promt:

```
ping 10.10.10.64
```

Ping (on the PC) should reply back with the ping time to the target. µC/TCP-IP target projects connected to the test PC on the same Ethernet switch or Ethernet cross-over cable achieve ping times of less than 2 milliseconds.

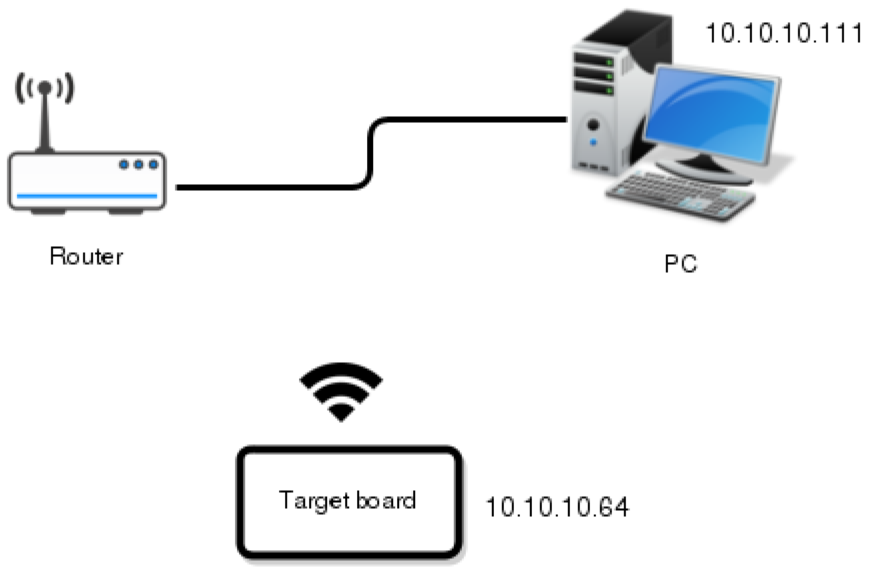

Figure - Sample Application Setup

After you have successfully completed and run the example project, you can use it as a starting point to run other µC/TCP-IP demos you may have purchased.

Adding Additional includes

Since the µC/TCP-IP module was added to the example project, the following include must be added to the app.c file :

```
#include <KAL/kal.h>
#include <Source/net.h>
#include <net_dev_????.h>
#include <net_wifi_mgr.h>
#include <net_dev_cfg.h>
#include <net_bsp.h>
```

Adding µC/TCP-IP application function

Before running the example application, you will need to add the new funtion, `AppInit_TCPIP()`, in your app.c file to initialize and setup the µC/TCP-IP stack. Section Tasks and Objects Initialization gives an example of the main application task inside which the `AppInit_TCPIP()` function will be called. Section WiFi Example Application gives an AppInit_TCPIP() example for a WiFi interface.

Those code examples will need to be modified in accordance with your project setup. For example, when adding an interface, your network device configuration object name (inside `net_dev_cfg.c`) will need to be specify and the IP address used in the example could need to be change to match your network.

Once the source code is built and loaded into the target, the target will respond to ICMP Echo (ping) requests.

Directories and Files

Below is a summary of all directories and files involved in the µC/TCP-IP stack. The '`<-Cfg`' on the far right indicates that these files are typically copied into the application (i.e., project) directory and edited based on project requirements.

```
\Micrium
    \Examples
            \<manufacturer>
                \<board_name>
                    \<project_name>
                        \<compiler>
                        \*.*
    \Software
        \uC-TCPIP
            \BSP
                \Template
                    \net_bsp_ether.c
                    \net_bsp_ether.h
                    \net_bsp_wifi.c
                    \net_bsp_wifi.h
            \Cfg
                \Template
                    \net_cfg.c                       <-Cfg
                    \net_cfg.h                       <-Cfg
                    \net_dev_cfg.c                   <-Cfg
                    \net_dev_cfg.h                   <-Cfg
            \Cmd
                \net_cmd.c
                \net_cmd.h
                \net_cmd_args_parser.c
                \net_cmd_args_parser.h
                \net_cmd_output.c
                \net_cmd_output.h
            \Dev
                \Ether
                    \<controller>
                        \net_dev_<controller>.c
                        \net_dev_<controller>.h
                    \PHY
                        \controller>
                            \net_phy_<controller>.c
                            \net_phy_<controller>.h
                        \Generic
                            \net_phy.c
                            \net_phy.h
                \WiFi
                    \<controller>
                        \net_dev_<controller>.c
                        \net_dev_<controller>.h
                    \Manager
                        \Generic
                            \net_wifi_mgr.c
                            \net_wifi_mgr.h
            \Examples
    \Init
     init_ether.c
     init_multiple_if.c
     init_wifi.c
                \Socket
                    tcp_client.c
```

```
                      tcp_server.c
                      udp_client.c
                      udp_server.c
       \TLS-SSL
        client_secure.c
        server_secure.c
      \IF
                  \net_if.c
                  \net_if.h
                  \net_if_802x.c
                  \net_if_802x.h
                  \net_if_ether.c
                  \net_if_ether.h
                  \net_if_wifi.c
                  \net_if_wifi.h
                  \net_if_loopback.c
                  \net_if_loopback.h
              \IP
                  \IPv4
                      \net_arp.c
                      \net_arp.h
                      \net_icmpv4.c
                      \net_imcpv4.h
                      \net_igmp.c
                      \net_igmp.h
                      \net_ipv4.c
                      \net_ipv4.h
                  \IPv6
                      \net_icmpv6.c
                      \net_icmpv6.h
                      \net_ipv6.c
                      \net_ipv6.h
                      \net_mldp.c
                      \net_mldp.h
                      \net_ndp.c
                      \net_ndp.h
              \Modules
                  \Common
                      \net_base64.c
                      \net_base64.h
                      \net_sha1.c
                      \net_sha1.h
              \Ports
                  \<architecture>
                      \<compiler>
                          \net_util_a.asm
              \Secure
                  net_secure.h
                  \<security_suite_name>
                      \net_secure_<suite_name>.c
                      \net_secure_<suite_name>.h
              \Source
                  \net.c
                  \net.h
                  \net_app.c
                  \net_app.h
                  \net_ascii.c
                  \net_ascii.h
                  \net.bsd.c
                  \net.bsd.h
                  \net.buf.c
                  \net.buf.h
                  \net_cache.c
                  \net_cache.h
                  \net_cfg_net.h
                  \net_conn.c
                  \net_conn.h
                  \net_ctr.c
                  \net_ctr.h
                  \net_def.h
                  \net_err.h
                  \net_icmp.c
                  \net_icmp.h
                  \net_ip.c
                  \net_ip.h
                  \net_mgr.c
                  \net_mgr.h
                  \net_sock.c
                  \net_sock.h
                  \net_stat.c
                  \net_stat.h
                  \net_tcp.c
                  \net_tcp.h
                  \net_tmr.c
                  \net_tmr.h
                  \net_type.h
                  \net_udp.c
                  \net_udp.h
                  \net_util.c
                  \net_util.h
```

`\Micrium`

Contains all software components and projects provided by Micrium.

`\Software`

This sub-directory contains all software components and projects.

`\uC-TCPIP`

This is the main directory for the μC/TCP-IP code.

BSP

\BSP\Template

This directory contains templates files, where functions might need to be implemented. See the section Network Board Support Package for further information.

Configuration

\Cfg

This directory contains configuration template file that must be copied to your project and modified following our requirements. See the section Configuration for further information.

Shell Commands

\Cmd

This directory contains the function that can be called from the command shell. This directory should be added to the project only if µC/Shell is present and network command must be added to this module.

See the following document for further information :

µC/Shell Documentation

Devices

Only the driver for your network controller(s) should be added to your project.

`\Dev`

This directory contains device drivers for different interfaces. Currently, µC/TCP-IP only supports one type of interface, Ethernet. µC/TCP-IP is tested with many types of Ethernet devices.

Ethernet

`\Ether`

Ethernet controller drivers are placed under the Ether sub-directory. Note that device drivers must also be called `net_dev_<controller>.*`.

`\<controller>`

The name of the Ethernet controller or chip manufacturer used in the project. The ‘<’ and ‘>’ are not part of the actual name. This directory contains the network device driver for the Network Controller specified.

`net_dev_<controller>.h` is the header file for the network device driver.

`net_dev_<controller>.c` contains C code for the network device driver API.

`\PHY`

This is the main directory for Ethernet Physical layer drivers.

`\Generic`

This is the directory for the Micrium provided generic PHY driver. Micrium's generic Ethernet PHY driver provides sufficient support for most (R)MII compliant Ethernet physical layer devices. A specific PHY driver may be developed in order to provide extended functionality such as link state interrupt support.

`net_phy.h` is the network physical layer header file.

`net_phy.c` provides the (R)MII interface port that is assumed to be part of the host Ethernet MAC. Therefore, (R)MII reads/writes *must* be performed through the network device API interface via calls to function pointers `Phy_RegRd()` and `Phy_RegWr()`.

Wireless

`\WiFi`

Wireless controller drivers are placed under the WiFi sub-directory. Note that device drivers must also be called net_dev_<controller>.*.

`\<controller>`

The name of the Wifi controller or chip manufacturer used in the project. The '<' and '>' are not part of the actual name. This directory contains the network device driver for the Network Controller specified.

`net_dev_<controller>.h` is the header file for the network device driver.

`net_dev_<controller>.c` contains C code for the network device driver API.

`\Manager`

This is the main directory for Wireless Manager layer.

`\Generic`

This is the directory for the Micriμm provided generic Wireless Manager layer. Micriμm's generic Wireless Manager layer provides sufficient support for most wireless devices that embed a wireless supplicant. A specific Wireless Manager may be developed in order to provide extended functionality.

`net_wifi_mgr.h` is the network Wireless Manager layer header file.

`net_wifi_mgr.c` provides functionality to access the device for management command that could required asynchronous response such as scan for available network.

Interface

This directory contains interface-specific files. Currently, μC/TCP-IP only supports three type of interfaces, Ethernet, wireless and loopback. The Ethernet and wireless interface-specific files are found in the following directories:

\IF

This is the main directory for network interfaces.

`net_if.*` presents a programming interface between higher μC/TCP-IP layers and the link layer protocols. These files also provide interface management routines to the application. This file should always be part of the project

`net_if_802x.*` contains common code to receive and transmit 802.3 and Ethernet packets. This file must not be modified. This file should always be part of the project

`net_if_ether.*` contains the Ethernet interface specifics. This file must not be modified and should be added to the project only if a Ethernet interface is used.

`net_if_wifi.*` contains the wireless interface specifics. This file must not be modified and should be added to the project only if a Wireless interface is used.

`net_if_loopback.*` contains loopback interface specifics. This file must not be modified and should be added to the project only if a Loopback interface is used.

File System Abstraction Layer

This directory contains the file system abstraction layer which allows the TCP-IP application such as µC/HTTPs, µC/FTPc, µC/FTPs, etc. with nearly any commercial or in-house file system. The abstraction layer for the selected file system is placed in a sub-directory under FS as follows:

`\FS`

This is the main FS directory that contain generic file system port header file. This file must be included if one or more application that required a file system such as µC/HTTPs, µC/FTPc, µC/FTPs, etc. are present in the project.

`\<file_system_name>`

This is the directory that contains the files to perform file system abstraction.

µC/TCP-IP has been tested with µC/FS-V4 and the file system layer files for this file system are found in the following directories:

`\Micrium\Software\uC-TCPIP\FS\uC-FS-V4\net_fs_v4.*`

Modules Code

This directory contains some code that can be shared between many Network application, such as HTTP, DNS, DHCP, etc. Each applications would tell you which file of this directory is required.

CPU Specific Code (Optimization)

Some functions can be optimized in assembly to improve the performance of the network protocol stack. An easy candidate is the checksum function. It is used at multiple levels in the stack, and a checksum is generally coded as a long loop.

`\Ports`

This is the main directory for processor specific code.

`\<architecture>`

The name of the CPU architecture that was ported to. The ‘<’ and ‘>’ are not part of the actual name.

`\<compiler>`

The name of the compiler or compiler manufacturer used to build code for the optimized function(s). The ‘<’ and ‘>’ are not part of the actual name.

`net_util_a.asm` contains assembly code for the specific CPU architecture. All functions that can be optimized for the CPU architecture are located here.

Core - CPU independent Source Code

This directory contains all the CPU and RTOS independent files for µC/TCP-IP. Nothing must be changed in this directory in order to use µC/TCP-IP.

`\Source`

This is the directory that contains all the CPU and RTOS independent source code files.

Examples Code

This directory contains code examples to help customers develop their network application. Those examples are given as guide lines and are not part of the µC/TCP-IP stack, therefore they are not part of the support Micriµm offers.

`\Examples`

This is the directory that contains the sample codes to help customers with their network application. It includes µC/TCP-IP stack initialization examples, socket programming examples, etc.

`\Init`

This is the directory that contains the sample codes to help customers with their network application. It includes μC/TCP-IP stack initialization examples.

`\Multicast`

This is the directory that contains the example codes to help customers with their network application. It includes multicast examples.

`\Socket`

This is the directory that contains the sample codes to help customers with their network application. It includes socket programming examples.

`\TLS-SSL`

This is the directory that contains the sample codes to help customers with their network application. It includes examples about how to use TLS/SSL with an application.

Notes

This section discusses the modules available for C/TCP-IP, and how they all fit together. A Windows®-based development platform is assumed. The directories and files make references to typical Windows-type directory structures. However, since C/TCP-IP is available in source form, it can also be used with any ANSI-C compatible compiler/linker and any Operating System.

The names of the files are shown in upper case to make them stand out. However, file names are actually lower case.

TCPIP Network Devices

The files in these directories are

```
\Micrium
 \Software
  \uC-TCPIP
   \Dev
    \Ether
     \PHY
      \Generic
     \<Controller>
    \WiFi
     \Manager
      \Generic
     \<Controller>
```

`\Micrium`

Contains all software components and projects provided by Micrium.

`\Software`

This sub-directory contains all software components and projects.

`\uC-TCPIP`

This is the main directory for the µC/TCP-IP code.

`\Dev`

This directory contains device drivers for different interfaces. Currently, µC/TCP-IP only supports one type of interface, Ethernet. µC/TCP-IP is tested with many types of Ethernet devices.

`\Ether`

Ethernet controller drivers are placed under the Ether sub-directory. Note that device drivers must also be called `net_dev_<controller>.*`.

`\WiFi`

Wireless controller drivers are placed under the WiFi sub-directory. Note that device drivers must also be called net_dev_<controller>.*.

`\PHY`

This is the main directory for Ethernet Physical layer drivers.

`\Generic`

This is the directory for the Micrium provided generic PHY driver. Micrium's generic Ethernet PHY driver provides sufficient support for most (R)MII compliant Ethernet physical layer devices. A specific PHY driver may be developed in order to provide extended functionality such as link state interrupt support.

`net_phy.h` is the network physical layer header file.

`net_phy.c` provides the (R)MII interface port that is assumed to be part of the host Ethernet MAC. Therefore, (R)MII reads/writes *must* be performed through the network device API interface via calls to function pointers `Phy_RegRd()` and `Phy_RegWr()`.

`\Manager`

This is the main directory for Wireless Manager layer.

`\Generic`

This is the directory for the Micriµm provided generic Wireless Manager layer. Micriµm's generic Wireless Manager layer provides sufficient support for most wireless devices that embed a wireless supplicant. A specific Wireless Manager may be developed in order to provide extended functionality.

`net_wifi_mgr.h` is the network Wireless Manager layer header file.

`net_wifi_mgr.c` provides functionality to access the device for management command that could required asynchronous response such as scan for available network.

`\<controller>`

The name of the Ethernet or wireless controller or chip manufacturer used in the project. The ‘<’ and ‘>’ are not part of the actual name. This directory contains the network device driver for the Network Controller specified.

`net_dev_<controller>.h` is the header file for the network device driver.

`net_dev_<controller>.c` contains C code for the network device driver API.

TCPIP Network Interface

This directory contains interface-specific files. Currently, µC/TCP-IP only supports three type of interfaces, Ethernet, wireless and loopback. The Ethernet and wireless interface-specific files are found in the following directories:

```
\Micrium
 \Software
  \uC-TCPIP
   \IF
```

`\Micrium`

Contains all software components and projects provided by Micrium.

`\Software`

This sub-directory contains all software components and projects.

`\uC-TCPIP`

This is the main µC/TCP-IP directory.

`\IF`

This is the main directory for network interfaces.

`net_if.*` presents a programming interface between higher µC/TCP-IP layers and the link layer protocols. These files also provide interface management routines to the application.

`net_if_802x.*` contains common code to receive and transmit 802.3 and Ethernet packets. This file should not need to be modified.

`net_if_ether.*` contains the Ethernet interface specifics. This file should not need to be modified.

`net_if_wifi.*` contains the wireless interface specifics. This file should not need to be modified.

`net_if_loopback.*` contains loopback interface specifics. This file should not need to be modified.

TCPIP Network File System Abstraction Layer

This directory contains the file system abstraction layer which allows the TCP-IP application such as µC/HTTPs, µC/FTPc, µC/FTPs, etc. with nearly any commercial or in-house file system. The abstraction layer for the selected file system is placed in a sub-directory under FS as follows:

```
\Micrium
 \Software
  \uC-TCPIP
   \FS
                \net_fs.h
    \<file_system_name>
```

`\Micrium`

Contains all software components and projects provided by Micrium.

`\Software`

This sub-directory contains all software components and projects.

`\uC-TCPIP`

This is the main µC/TCP-IP directory.

`\FS`

This is the main FS directory that contain generic file system port header file. This file must be included if one or more application that required a file system such as µC/HTTPs, µC/FTPc, µC/FTPs, etc. are present in the project.

`\<file_system_name>`

This is the directory that contains the files to perform file system abstraction.

µC/TCP-IP has been tested with µC/FS-V4 and the file system layer files for this file system are found in the following directories:

```
\Micrium\Software\uC-TCPIP\FS\uC-FS-V4\net_fs_v4.*
```

TCPIP Network CPU Specific Code

Some functions can be optimized in assembly to improve the performance of the network protocol stack. An easy candidate is the checksum function. It is used at multiple levels in the stack, and a checksum is generally coded as a long loop.

```
\Micrium
 \Software
  \uC-TCPIP
   \Ports
    \<architecture>
     \<compiler>
      \net_util_a.asm
```

`\Micrium`

Contains all software components and projects provided by Micrium.

`\Software`

This sub-directory contains all software components and projects.

`\uC-TCPIP`

This is the main µC/TCP-IP directory.

`\Ports`

This is the main directory for processor specific code.

`\<architecture>`

The name of the CPU architecture that was ported to. The ‘<’ and ‘>’ are not part of the actual name.

`\<compiler>`

The name of the compiler or compiler manufacturer used to build code for the optimized function(s). The ‘<’ and ‘>’ are not part of the actual name.

`net_util_a.asm` contains assembly code for the specific CPU architecture. All functions that can be optimized for the CPU architecture are located here.

TCPIP Network CPU Independent Source Code

This directory contains all the CPU and RTOS independent files for μC/TCP-IP. Nothing should be changed in this directory in order to use μC/TCP-IP.

```
\Micrium
 \Software
  \uC-TCPIP
   \Source
```

`\Micrium`

Contains all software components and projects provided by Micrium.

`\Software`

This sub-directory contains all software components and projects.

`\uC-TCPIP`

This is the main μC/TCP-IP directory.

`\Source`

This is the directory that contains all the CPU and RTOS independent source code files.

TCPIP Network Security Manager CPU Independent Source Code

This directory contains all the CPU independent files for µC/TCP-IP Network Security Manager. Nothing should be changed in this directory in order to use µC/TCP-IP.

```
\Micrium
 \Software
  \uC-TCPIP
   \Secure
    \<security_suite_name>
```

`\Micrium`

Contains all software components and projects provided by Micrium.

`\Software`

This sub-directory contains all software components and projects.

`\uC-TCPIP`

This is the main µC/TCP-IP directory.

`\Secure`

This is the main Secure directory that contains the generic secure port header file. This file should be included in the project only if a security suite is available and is to be used by the application.

`\Secure\<security_suite_name>`

This is the directory that contains the files to perform security suite abstraction. These files should only be included in the project if a security suite (i.e Mocana - NanoSSL) is available and is to be used by the application.

TCPIP Network Examples Code

This directory contains code examples to help customers develop their network application. Those examples are given as guide lines and are not part of the µC/TCP-IP stack, therefore they are not part of the support Micriµm offers.

```
\Micrium
 \Software
  \uC-TCPIP
   \Examples
                \Init
                \Socket
                \TLS-SSL
```

`\Micrium`

Contains all software components and projects provided by Micriµm.

`\Software`

This sub-directory contains all software components and projects.

`\uC-TCPIP`

This is the main µC/TCP-IP directory.

`\Examples`

This is the directory that contains the sample codes to help customers with their network application. It includes µC/TCP-IP stack initialization examples, socket programming examples, etc.

`\Init`

This is the directory that contains the sample codes to help customers with their network application. It includes µC/TCP-IP stack initialization examples.

`\Multicast`

This is the directory that contains the example codes to help customers with their network application. It includes multicast examples.

`\Socket`

This is the directory that contains the sample codes to help customers with their network application. It includes socket programming examples.

`\TLS-SSL`

This is the directory that contains the sample codes to help customers with their network application. It includes examples about how to use TLS/SSL with an application.

Configuration

Prior to usage, µC/TCP-IP must be properly configured. There are four groups of configuration parameters:

- Network Stack Configuration
- Network Tasks Configuration
- Network Interface Configuration
- LIB Memory Heap Configuration

This chapter explains how to setup all these groups of configuration.

Network Stack Configuration

µC/TCP-IP is configurable at compile time via approximately 50 `#defines` in the application's `net_cfg.h` file. µC/TCP-IP uses `#defines` because they allow code and data sizes to be scaled at compile time based on enabled features and the configured number of network objects. This allows the ROM and RAM footprints of µC/TCP-IP to be adjusted based on application requirements.

Most of the `#defines` should be configured with the default configuration values. A handful of values may likely never change because there is currently only one configuration choice available. This leaves approximately a dozen values that should be configured with values that may deviate from the default configuration.

It is recommended that the configuration process begins with the default configuration values which in the next sections will be shown in **bold**.

The sections in this chapter are organized following the order in µC/TCP-IP's template configuration file, `net_cfg.h`.

Sub-modules Configuration

µC/TCP-IP contains code that can use sub-modules such as DNS client to perform some specific operation and extend the functionalities of some particular API.

Constant	Description	Possible Values
NET_EXT_MODULE_CFG_DNS_EN	Select portions of µC/TCP-IP code may call µC/DNSc API to resolve remote hostname. If µC/DNSc files/functions are included in the µC/TCP-IP build set NET_EXT_MODULE_CFG_DNS_EN to `DEF_ENABLED`. Set to `DEF_DISABLED` otherwise.	`DEF_ENABLED` or **`DEF_DISABLED`**

Table - Compile Feature Constants

Task Queue Configuration

The µC/TCP-IP stack has two queues that need to be configured. The first one is the Rx queue and is used to store the Rx buffer that have been filled and are ready to be process. The second queue is the Tx deallocation and is used to store the Tx buffers that are ready to be deallocate.

Constant	Description	Possible Values
NET_CFG_IF_RX_Q_SIZE	Should be configured such that it reflects the total number of receive buffer minus the number of receive descriptor on all physical interfaces. If DMA is not available, or a combination of DMA and I/O based interfaces are configured then this number reflects the maximum number of packets that can be acknowledged and signaled during a single receive interrupt event for all interfaces.	Depends on the device configuraiton (see net_dev_cfg.c)
NET_CFG_IF_TX_DEALLOC_Q_SIZE	Should be defined to be the total number of small and large transmit buffers declared for all interfaces.	Depends on the device configuraiton (see net_dev_cfg.c)

Table - Task Queue Constants

Compile Features Configuration

Constant	Description	Possible Values
NET_CFG_OPTIMIZE_ASM_EN	Select portions of µC/TCP-IP code may call optimized assembly functions by configuring `NET_CFG_OPTIMIZE_ASM_EN` to `DEF_ENABLED.` If optimized assembly files/functions are included in the µC/TCP-IP build set `NET_CFG_OPTIMIZE_ASM_EN` to `DEF_NEABLED`. Set to `DEF_DISABLED` otherwise.	`DEF_ENABLED` or **`DEF_DISABLED`**

Table - Compile Feature Constants

Debug Features Configuration

Constant	Description	Possible Values
NET_DBG_CFG_MEM_CLR_EN	Is used to clear internal network data structures when allocated or de-allocated. By clearing, all bytes in internal data structures are set to '0' or to default initialization values. This configuration is typically set it to `DEF_DISABLED` unless the contents of the internal network data structures need to be examined for debugging purposes. Having the internal network data structures cleared generally helps to differentiate between "proper" data and "pollution".	`DEF_ENABLED` or **`DEF_DISABLED`**

Table - Debug Feature Constants

Argument Check Configuration

Most functions in µC/TCP-IP include code to validate arguments that are passed to it. Specifically, µC/TCP-IP checks to see if passed pointers are `NULL`, if arguments are within valid ranges, etc. The following constants configure additional argument checking.

Constant	Description	Possible Values
NET_ERR_CFG_ARG_CHK_EXT_EN	Allows code generated to check arguments for functions that can be called by the user and, for functions which are internal but receive arguments from an API that the user can call. Also, enabling this check verifies that µC/TCP-IP is initialized before API tasks and functions perform the desired function.	`DEF_ENABLED` or **`DEF_DISABLED`**
NET_ERR_CFG_ARG_CHK_DBG_EN	Allows code to be generated which checks to make sure that pointers passed to functions are not NULL, and that arguments are within range, etc.	`DEF_ENABLED` or **`DEF_DISABLED`**

Table - Argument Check Constants

Counters Configuration

µC/TCP-IP contains code that increments counters to keep track of statistics such as the number of packets received, the number of packets transmitted, etc. Also, µC/TCP-IP contains counters that are incremented when error conditions are detected.

Constant	Description	Possible Values
NET_CTR_CFG_STAT_EN	Determines whether the code and data space used to keep track of statistics will be included.	`DEF_ENABLED` or **`DEF_DISABLED`**
NET_CTR_CFG_ERR_EN	Determines whether the code and data space used to keep track of errors will be included.	`DEF_ENABLED` or **`DEF_DISABLED`**

Table - Counter Management Constants

Timer Configuration

μC/TCP-IP manages software timers used to keep track of timeouts and execute callback functions when needed.

Constant	Description	Possible Values
NET_TMR_CFG_NBR_TMR	Determines the number of timers that μC/TCP-IP will be managing. Number of timer to configure depends on the network application. It is recommended to set a large number of timer and look at the counter *NetTmr_PoolStat* to see the maximum number of timer needed and make sure that we never run out of free buffer by looking at the error counter *Net_ErrCtrs.Tmr.NoneAvailCtr.*	Depends on TCPIP stack configuration.
NET_TMR_CFG_TASK_FREQ	Determines how often (in Hz) network timers are to be updated. This value *must not* be configured as a floating-point number.	Typically set to **`10`** `Hz`

Table - Timer Management Constants

Network Interfaces Configuration

Constant	**Description**	**Possible Values**
NET_IF_CFG_MAX_NBR_IF	Determines the maximum number of network interfaces that µC/TCP-IP may create at run-time.	**1u** if a single network interface is present.
NET_IF_CFG_LOOPBACK_EN	Determines whether the code and data space used to support the loopback interface for internal-only communication only will be included.	DEF_ENABLED or **DEF_DISABLED**
NET_IF_CFG_ETHER_EN	Determines whether the code and data space used to support Ethernet interfaces and devices will be included.	**DEF_ENABLED** or DEF_DISABLED
NET_IF_CFG_WIFI_EN	Determines whether the code and data space used to support wireless interfaces and devices will be included.	DEF_ENABLED or **DEF_DISABLED**
NET_IF_CFG_TX_SUSPEND_TIMEOUT_MS	Configures the network interface transmit suspend timeout value. The value is specified in integer milliseconds.	**1u**

Table - Interfaces Configuration Constants

Address Resolution Protocol (ARP) Configuration

ARP is only used when the IPv4 stack is enabled.

Constant	**Description**	**Possible Values**
NET_ARP_CFG_CACHE_NBR	Configures the number of ARP cache entries.	3u

Table - ARP Configuration Constants

ARP caches the mapping of IPv4 addresses to physical (i.e., MAC) addresses. `NET_ARP_CFG_NBR_CACHE` configures the number of ARP cache entries. Each cache entry requires approximately `18` bytes of RAM, plus seven pointers, plus a hardware address and protocol address (`10` bytes assuming Ethernet interfaces and IPv4 addresses).

The number of ARP caches required by the application depends on how many different hosts are expected to communicate. If the application *only* communicates with hosts on remote networks via the local network's default gateway (i.e., router), then only a single ARP cache needs to be configured.

To test µC/TCP-IP with a smaller network, a default number of 3 ARP caches should be sufficient.

Neighbor Discovery Protocol (NDP) Configuration

NDP is only used when the IPv6 stack is enabled.

Constant	Description	Possible Values
NET_NDP_CFG_CACHE_NBR	Configures the number of NDP Neighbor cache entries.	6u
NET_NDP_CFG_DEST_NBR	Configures the number of NDP Destination cache entries.	5u
NET_NDP_CFG_PREFIX_NBR	Configures the number of NDP Prefix entries.	5u
NET_NDP_CFG_ROUTER_NBR	Configures the number of NDP Router entries.	1u

Table - NDP Configuration Constants

NDP caches the mapping of IPv6 addresses to physical (i.e., MAC) addresses. `NET_NDP_CFG_NBR_CACHE` configures the number of NDP Neighbor cache entries. Each cache entry requires approximately `18` bytes of RAM, plus seven pointers, plus a hardware address and protocol address (`22` bytes assuming Ethernet interfaces and IPv6 addresses).

NDP also caches recent IPv6 destination addresses by mapping next-hop address to final destination address. It allows the µC/TCP-IP stack not having to re-calculating the next-hop for each packet to send. `NET_NDP_CFG_DEST_NBR` configured the numver of NDP destination caches available for the TCPIP stack.

In IPv6, routers send router advertisement messages to inform hosts on different values like the IPv6 prefix considered on-link. Those on-link prefix are stored in a NDP prefix list. `NET_NDP_CFG_PREFIX_NBR` configured the the number of prefix entries available in the list.

IPv6 defines an algorithm to chose the adequate router on the network to transmit packet outside in case more than one IPv6 router is present. `NET_NDP_CFG_ROUTER_NBR` defines the number of router information that can be store by the µC/TCP-IP stack.

IPv4 Layer Configuration

Constant	Description	Possible Values
NET_IPv4_CFG_EN	Enables the IPv4 module.	**DEF_ENABLED** or DEF_DISABLED
NET_IPv4_CFG_IF_MAX_NBR_ADDR	Determines the maximum number of IPv4 addresses that may be configured per network interface at run-time.	At least **1**

Table - IPv4 Configuration Constants

IPv6 Layer Configuration

Constant	Description	Possible Values
NET_IPv6_CFG_EN	Enables the IPv6 module.	**DEF_ENABLED** or DEF_DISABLED
NET_IPv6_CFG_ADDR_AUTO_CFG_EN	Enables the IPv6 Staless Address Auto-Configuration module.	**DEF_ENABLED** or DEF_DISABLED
NET_IPv6_CFG_DAD_EN	Enables the Duplication Address Detection (DAD) module.	**DEF_ENABLED** or DEF_DISABLED
NET_IPv6_CFG_IF_MAX_NBR_ADDR	Determines the maximum number of IPv6 addresses that may be configured per network interface at run-time.	At least **2**

Table - IPv6 Configuration Constants

Multicast Configuration (IGMP and MLDP)

Constant	Description	Possible Values
NET_MCAST_CFG_IPv4_RX_EN	Enables the multicast support in reception for IPv4.	**`DEF_ENABLED`** or `DEF_DISABLED`
NET_MCAST_CFG_IPv4_TX_EN	Enables the multicast support in transmittion for IPv4.	**`DEF_ENABLED`** or `DEF_DISABLED`
NET_MCAST_CFG_HOST_GRP_NBR_MAX	Configures the maximum number of IGMP host groups that may be joined at any one time.	2u

Table - IGMP Configuration Constants

`NET_IGMP_CFG_MAX_NBR_HOST_GRP` configures the maximum number of IGMP host groups that may be joined at any one time. Each group entry requires approximately `12` bytes of RAM, plus three pointers, plus a protocol address (`4` bytes assuming IPv4 address).

The number of IGMP host groups required by the application depends on how many host groups are expected to be joined at a given time. Since each configured multicast address requires its own IGMP host group, it is recommended to configure at least one host group per multicast address used by the application, plus one additional host group. Thus for a single multicast address, it is recommended to set `NET_IGMP_CFG_MAX_NBR_HOST_GRP` with an initial value of `2`.

Socket Layer Configuration

μC/TCP-IP supports BSD 4.x sockets and basic socket API for the TCP/UDP/IP protocols.

Constant	Description	Possible Values
NET_SOCK_CFG_SOCK_NBR_TCP	Configures total number of TCP connections.	5
NET_SOCK_CFG_SOCK_NBR_UDP	Configures total number of UDP connections.	2
NET_SOCK_CFG_SEL_EN	Configures socket select functionality.	**DEF_ENABLED** or DEF_DISABLED
NET_SOCK_CFG_CONN_ACCEPT_Q_SIZE_MAX	Configures stream-type sockets' accept queue.	2
NET_SOCK_CFG_RX_Q_SIZE_OCTET	Configurse socket receive queue buffer size.	4096
NET_SOCK_CFG_TX_Q_SIZE_OCTET	Configures socket transmit queue buffer size.	4096

Table - Socket Configuraiton Constants

See Configuring window sizes for more information about how to configure receive and transmit queues buffer size.

TCP Layer Configuration

Constant	Description	Possible Values
NET_TCP_CFG_EN	Enables the TCP module.	**DEF_ENABLED** or DEF_DISABLED

Table - TCP Configuration Constants

UDP Layer Configuration

Constant	Description	Possible Values
NET_UDP_CFG_RX_CHK_SUM_DISCARD_EN	Is used to determine whether received UDP packets without a valid checksum are discarded or are handled and processed. Before a UDP Datagram Check-Sum is validated, it is necessary to check whether the UDP datagram was transmitted with or without a computed Check-Sum.	DEF_ENABLED or **DEF_DISABLED**
NET_UDP_CFG_TX_CHK_SUM_EN	Is used to determine whether UDP checksums are computed for transmission to other hosts.	DEF_ENABLED or **DEF_DISABLED**

Table - UDP Configuration Constants

Transport Layer Security Configuration

Constant	Description	Possible Values
NET_SECURE_CFG_EN	Configures network security manager.	`DEF_ENABLED` or **`DEF_DISABLED`**
NET_SECURE_CFG_MAX_NBR_SOCK_SERVER	Configures total number of server secure sockets.	5
NET_SECURE_CFG_MAX_NBR_SOCK_CLIENT	Configures total number of client secure sockets.	5
NET_SECURE_CFG_MAX_CERT_LEN	Configures max length (in octets) of Server certificates.	1500
NET_SECURE_CFG_MAX_KEY_LEN	Configures max length (in octets) of Server keys.	1500
NET_SECURE_CFG_MAX_NBR_CA	Configures maximum number of certificate authorities that can be installed.	1
NET_SECURE_CFG_MAX_CA_CERT_LEN	Configure maximum length (in octets) of certificate authority certificates.	1500

Table - Security Management Constants

Network Tasks Configuration

This section defines the configuration structures related to C/TCP-IP but that are application-specific. All these configurations relate to the RTOS. For many OSs, the C/TCP-IP task priorities and stack sizes will need to be explicitly configured for the particular OS (consult the specific OS's documentation for more information).

These configurations are defined in the `net_cfg.c` file.

Network Task Configuration

µC/TCP-IP use the following structure to configure its network tasks.

```
typedef  struct  net_task_cfg {
    CPU_INT32U   Prio;                                      /* Task priority.
*/
    CPU_INT32U   StkSizeBytes;                              /* Size of the stack.
*/
    void        *StkPtr;                                    /* Pointer to base of the stack.
*/
} NET_TASK_CFG;
```

Listing - µC/TCP-IP task configuration structure

µC/TCP-IP stack has three internal tasks that need to be configured : the Receive task, the Transmit De-allocation task and the Timer task. Each task has its own `NET_TASK_CFG` object defining the task priority, the task's stack size and the pointer to start of task stack.

Task Priorities

We recommend you configure the Network Protocol Stack task priorities & Network application task priorities as follows:

- Network TX De-allocation task (highest priority)
- Network application tasks, such as HTTPs instance.

- Network timer task
- Network RX task (lowest priority)

We recommend that the uC/TCP-IP Timer task and network interface Receive task to be lower priority than almost all other application tasks; but we recommend that the network interface Transmit De-allocation task to be higher priority than all application tasks that use uC/TCP-IP network services.

However better performance can be observed when the network application task is set with the lowest priority. Some experimentation could be required to identify the best task priority configuration.

Task Stack Size

In general, the size of µC/TCP-IP task stacks is dependent on the CPU architecture and compiler used.

The only guaranteed method of determining the required task stack sizes is to calculate the maximum stack usage for each task. Obviously, the maximum stack usage for a task is the total stack usage along the task’s most-stack-greedy function path plus the (maximum) stack usage for interrupts. Note that the most-stack-greedy function path is not necessarily the longest or deepest function path.

The easiest and best method for calculating the maximum stack usage for any task/function should be performed statically by the compiler or by a static analysis tool since these can calculate function/task maximum stack usage based on the compiler’s actual code generation and optimization settings. So for optimal task stack configuration, we recommend to invest in a task stack calculator tool compatible with your build toolchain.

On ARM processors, experience has shown that configuring the task stacks to 1024 `OS_STK` entries (4,096 bytes) is sufficient for most applications. Certainly, the stack sizes may be examined and reduced accordingly once the run-time behavior of the device has been analyzed and additional stack space deemed to be unnecessary.

Task Stack Location and Allocation

If a specific memory location is desired for a task stack, the `StkPtr` parameter can be set to point to this specific memory segment. Else, if `StkPtr` is set to NULL, the task stack will be allocate on µC/LIB Heap.

Network Interface Configuration

This section gives more details on how to configure a network interface for C/TCP-IP.

Buffers' Management

This section describe how µC/TCP-IP uses buffers to receive and transmit application data and network protocol control information. You should understand how network buffers are used by µC/TCP-IP to correctly configure your interface(s).

Network Buffers

µC/TCP-IP stores transmitted and received data in data structures known as Network Buffers. Each Network Buffer consists of two parts: the Network Buffer header and the Network Buffer Data Area pointer. Network Buffer headers contain information about the data pointed to via the data area pointer. Data to be received or transmitted is stored in the Network Buffer Data Area.

µC/TCP-IP is designed with the inherent constraints of an embedded system in mind, the most important being the restricted RAM space. µC/TCP-IP defines network buffers for the Maximum Transmission Unit (MTU) of the Data Link technology used, which is most of the time Ethernet. Default Ethernet’s maximum transmit unit (MTU) size is 1500 bytes.

Receive Buffers

Network Buffers used for reception for a Data Link technology are buffers that can hold one maximum frame size. Because it is impossible to predict how much data will be received, only large buffers can be configured. Even if the packet does not contain any payload, a large buffer must be used, as worst case must always be assumed.

Transmit Buffers

On transmission, the number of bytes to transmit is always known, so it is possible to use a Network Buffer size smaller than the maximum frame size. µC/TCP-IP allows you to reduce the RAM usage of the system by defining small buffers. When the application does not require a full size frame to transmit, it is possible to use smaller Network Buffers. Depending on the configuration, up to eight pools of Network Buffer related objects may be created per network interface. Only four pools are shown below and the remaining pools are used for maintaining Network Buffer usage statistics for each of the pools shown.

In transmission, the situation is different. The TCP/IP stack knows how much data is being transmitted. In addition to RAM being limited in embedded systems, another feature is the small amount of data that needs to be transmitted. For example, in the case of sensor data to be transmitted periodically, a few hundred bytes every second can be transferred. In this case, a small buffer can be used and save RAM instead of waste a large transmit buffer. Another example is the transmission of TCP acknowledgment packets, especially when they are not carrying any data back to the transmitter. These packets are also small and do not require a large transmit buffer. RAM is also saved.

µC/TCP-IP requires that network buffer sizes configured in `net_dev_cfg.c` satisfy the minimum and maximum packet frame sizes of network interfaces/devices.

Assuming an Ethernet interface (with non-jumbo or VLAN-tagged frames), the minimum frame packet size is 64 bytes (including its 4-byte CRC). If an Ethernet frame is created such that the frame length is less than 60 bytes (before its 4-byte CRC is appended), frame padding must be appended by the network driver or the Ethernet network interface layer to the application data area to meet Ethernet’s minimum packet size. For example, the ARP protocol typically creates packets of 42 bytes and therefore 18 bytes of padding must be added. The additional padding must fit within the network buffer’s data area.

Ethernet’s maximum transmit unit (MTU) size is 1500 bytes. When TCP is used as the transport protocol, TCP and IP protocol header sizes are subtracted from Ethernet’s 1500-byte MTU. A maximum of 1460 bytes of TCP application data may be sent in a full-sized Ethernet frame.

In addition, the variable size of network packet protocol headers must also be considered when configuring buffer sizes. The following computations demonstrate how to configure network buffer sizes to transmit and receive maximum sized network packets.

Typical Buffers Size

The following table shows how each network buffer should be configured to handle the majority of worst cases.

Type of network buffer	Size
Receive Large Buffer	1518 + Alignment
Transmit Large Buffer	1518 + Alignment
Transmit Small Buffer	64 + Alignment

Network Device Configuration

All C/TCP-IP device drivers require a configuration structure for each device that must be compiled into your driver. You must place all device configuration structures and declarations within a pair of files named `net_dev_cfg.c` and `net_dev_cfg.h`.

Micriµm provides sample configuration code free of charge; however, most sample code will likely require modification depending on the combination of compiler, processor, evaluation board, and device hardware used.

Memory Configuration

The first step in creating a device driver configuration for µC/TCP-IP begins with the memory configuration structure. This section describes the memory configuration settings for most device drivers, and should provide you an in-depth understanding of memory configuration. You will also discover which settings to modify in order to enhance the performances of the driver.

The listing below shows a sample memory configuration structure.

```
const  NET_DEV_CFG  NetDev_Cfg_Dev1 = {
                           /* Structure member:                                        */
    NET_IF_MEM_TYPE_MAIN,  /*     .RxBufPoolType                                       */ (1)
    1518u,                 /*     .RxBufLargeSize                                      */ (2)
       9u,                 /*     .RxBufLargeNbr                                       */ (3)
      16u,                 /*     .RxBufAlignOctets                                    */ (4)
       0u,                 /*     .RxBufIxOffset                                       */ (5)

    NET_IF_MEM_TYPE_MAIN,  /*     .TxBufPoolType                                       */ (6)
    1606u,                 /*     .TxBufLargeSize                                      */ (7)
       4u,                 /*     .TxBufLargeNbr                                       */ (8)
     256u,                 /*     .TxBufSmallSize                                      */ (9)
       2u,                 /*     .TxBufSmallNbr                                       */ (10)
      16u,                 /*     .TxBufAlignOctets                                    */ (11)
       0u,                 /*     .TxBufIxOffset                                       */ (12)

    0x00000000u,           /*     .MemAddr                                             */ (13)
             0u,           /*     .MemSize                                             */ (14)

    NET_DEV_CFG_FLAG_NONE, /*     .Flag                                                */ (15)
};
```

Listing - Sample memory configuration

1. `.RxBufPoolType` specifies the memory location for the receive data buffers. Buffers may located either in main memory or in a dedicated memory region. This setting is used by the IF layer to initialize the Rx memory pool. This field must be set to one of two macros: `NET_IF_MEM_TYPE_MAIN` or `NET_IF_MEM_TYPE_DEDICATED`. You may want to set this field when DMA with dedicated memory is used. It is possible that you might have to store descriptors within the dedicated memory if your device requires it.

2. `.RxBufLargeSize` specifies the size of all receive buffers. Specifying a value is required. The buffer length is set to 1518 bytes which corresponds to the Maximum Transmission Unit (MTU) of an Ethernet network. For DMA-based Ethernet controllers, you must set the receive data buffer size to be greater or equal to the size of the largest receivable frame. If the size of the total buffer allocation is greater than the amount of available memory in the chosen memory region, a run-time error will be generated when the device is initialized.

3. `.RxBufLargeNbr` specifies the number of receive buffers that will be allocated to the device. There should be at least one receive buffer allocated, and it is recommended to have at least ten receive buffers. The optimal number of receive buffers depends on your application.

4. `.RxBufAlignOctets` specifies the required alignment of the receive buffers, in bytes. Some devices require that the receive buffers be aligned to a specific byte boundary. Additionally, some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it is probably a best practice to align buffers to the data bus width of the processor, which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a four-byte boundary.

5. `.RxBufIxOffset` specifies the receive buffer offset in bytes. Most devices receive packets starting at base index zero in the network buffer data areas. However, some devices may buffer additional bytes prior to the actual received Ethernet packet. This setting configures an offset to ignore these additional bytes. If a device does not buffer any additional bytes ahead of the received Ethernet packet, then an offset of 0 must be specified. However, if a device does buffer additional bytes ahead of the received Ethernet packet, then you should configure this offset with the number of additional bytes. Also, the receive buffer size must also be adjusted by the number of additional bytes.

6. `.TxBufPoolType` specifies the memory placement of the transmit data buffers. Buffers may be placed either in main memory or in a dedicated memory region. This field is used by the IF layer, and it should be set to one of two macros: `NET_IF_MEM_TYPE_MAIN` or `NET_IF_MEM_TYPE_DEDICATED`. When DMA descriptors are used, they may be stored into the dedicated memory.

7. `.TxBufLargeSize` specifies the size of the large transmit buffers in bytes. This field has no effect if the number of large transmit buffers is configured to zero. Setting the size of the large transmit buffers below 1594 bytes may hinder the µC/TCP-IP module’s ability to transmit full sized IP datagrams since IP transmit fragmentation is not yet supported. We recommend setting this field between 1594 and 1614 bytes in order to accommodate all of the maximum transmit packet sizes for C/TCP-IP’s protocols.

   You can optimize the transmit buffer if you know in advance what the maximum size of the packets the user will want to transmit through the device are.

8. `.TxBufLargeNbr` specifies the number of large transmit buffers allocated to the device. You may set this field to zero to make room for additional small transmit buffers, however, the size of the maximum transmittable packet will then depend on the size of the small transmit buffers.

9. `.TxBufSmallSize` specifies the small transmit buffer size. For devices with a minimal amount of RAM, it is possible to allocate small transmit buffers as well as large transmit buffers. In general, we recommend a 152 byte small transmit buffer size, however, you may adjust this value according to the application requirements. This field has no effect if the number of small transmit buffers is configured to zero.

10. `.TxBufSmallNbr` specifies the numbers of small transmit buffers. This field controls the number of small transmit buffers allocated to the device. You may set this field to zero to make room for additional large transmit buffers if required.

11. `.TxBufAlignOctets` specifies the transmit buffer alignment in bytes. Some devices require that the transmit buffers be aligned to a specific byte boundary. Additionally, some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it's probably a best practice to align buffers to the data bus width of the processor which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a four-byte boundary.

12. `.TxBufIxOffset` specifies the transmit buffer offset in bytes. Most devices only need to transmit the actual Ethernet packets as prepared by the higher network layers. However, some devices may need to transmit additional bytes prior to the actual Ethernet packet. This setting configures an offset to prepare space for these additional bytes. If a device does not transmit any additional bytes ahead of the Ethernet packet, the default offset of zero should be configured. However, if a device does transmit additional bytes ahead of the Ethernet packet then configure this offset with the number of additional bytes. The transmit buffer size must also be adjusted to include the number of additional bytes.

13. `.MemAddr` specifies the starting address of the dedicated memory region for devices with

this memory type. For devices with non-dedicated memory, you can initialize this field to zero. You may use this setting to put DMA descriptors into the dedicated memory.

14. `.MemSize` specifies the size of the dedicated memory region in bytes for devices with this memory type. For devices with non-dedicated memory, you can initialize this field to zero. You may use this setting to put DMA descriptors into the dedicated memory.

15. `.Flags` specify the optional configuration flags. Configure (optional) device features by logically OR'ing bit-field flags:
    NET_DEV_CFG_FLAG_NONE No device configuration flags selected.
    NET_DEV_CFG_FLAG_SWAP_OCTETS Swap data bytes (i.e., swap data words' high-order bytes with data words' low-order bytes, and vice-versa) if required by device-to-CPU data bus wiring and/or CPU endian word order.

Ethernet Device Configuration

Listing - Ethernet Device Configuration shows a sample Ethernet configuration structure for Ethernet devices.

```
const  NET_DEV_CFG_ETHER  NetDev_Cfg_Dev1_0 = {
                           /* Structure member:                                         */
    NET_IF_MEM_TYPE_MAIN,  /*     .RxBufPoolType                                        */ (1)
    1518u,                 /*     .RxBufLargeSize                                       */
      10u,                 /*     .RxBufLargeNbr                                        */
       4u,                 /*     .RxBufAlignOctets                                     */
       0u,                 /*     .RxBufIxOffset                                        */

    NET_IF_MEM_TYPE_MAIN,  /*     .TxBufPoolType                                        */
    1606u,                 /*     .TxBufLargeSize                                       */
       4u,                 /*     .TxBufLargeNbr                                        */
     152u,                 /*     .TxBufSmallSize                                       */
       4u,                 /*     .TxBufSmallNbr                                        */
       4u,                 /*     .TxBufAlignOctets                                     */
       0u,                 /*     .TxBufIxOffset                                        */

    0x00000000u,           /*     .MemAddr                                              */
             0u,           /*     .MemSize                                              */

    NET_DEV_CFG_FLAG_NONE, /*     .Flag                                                 */

             6u,           /*     .RxDescNbr                                             */ (2)
             6u,           /*     .TxDescNbr                                             */ (3)
    0x40028000u,           /*     .BaseAddr                                              */ (4)
             0u,           /*     .DataBusSizeNbrBits                                    */ (5)
   "00:50:C2:25:61:00",    /*     .HW_AddrStr                                            */ (6)
};
```

Listing - Ethernet Device Configuration

1. Memory configuration of the Ethernet Device. See “Memory Configuration” for further information about how to configure the memory of your Ethernet interface.

2. `.RxDescNbr` specifies the number of receive descriptors. For DMA-based devices, this value is used by the device driver during initialization in order to allocate a fixed-size pool of receive descriptors to be used by the device. The number of descriptors must be less than the number of configured receive buffers. We recommend setting this value to something within 40% and 70% of the number of receive buffers. Non-DMA based devices may configure this value to zero. You must use this setting with DMA based devices and at least two descriptors must be set.

3. `.TxDescNbr` specifies the number of transmit descriptors. For DMA based devices, this value is used by the device driver during initialization to allocate a fixed size pool of transmit descriptors to be used by the device. For best performance, it's recommended to set the number of transmit descriptors equal to the number of small, plus the number of large transmit buffers configured for the device. Non-DMA based devices may configure this value to zero. You must use this setting with DMA based devices and set at least two descriptors.

4. `.BaseAddr` specifies the base address of the device's hardware/registers.

5. `.DataBusSizeNbrBits` specifies the size of device's data bus (in bits), if available.

6. `.HW_AddrStr` specifies the desired device hardware address; may be NULL address or string if the device hardware address is configured or set at run-time.Depending on the driver, if this value is kept NULL or invalid, most of the device driver will automatically try to load and use the hardware address located in the memory of the device.

Ethernet PHY Configuration

Listing - Ethernet PHY Configuration shows a typical Ethernet PHY configuration structure.

```
NET_PHY_CFG_ETHER NetPhy_Cfg_FEC_0= {
  NET_PHY_ADDR_AUTO,                    (1)
  NET_PHY_BUS_MODE_MII,                 (2)
  NET_PHY_TYPE_EXT                      (3)
  NET_PHY_SPD_AUTO,                     (4)
  NET_PHY_DUPLEX_AUTO,                  (5)
};
```

Listing - Ethernet PHY Configuration

1. PHY Address. This field represents the address of the PHY on the (R)MII bus. The value configured depends on the PHY and the state of the PHY pins during power-up. Developers may need to consult the schematics for their board to determine the configured PHY address. Alternatively, the PHY address may be detected automatically by specifying `NET_PHY_ADDR_AUTO`; however, this will increase the initialization latency

of μC/TCP-IP and possibly the rest of the application depending on where the application places the call to `NetIF_Start()`.

2. `PHY bus mode. This value should be set to one of the following values depending on the hardware capabilities and schematics of the development board. The network device BSP should configure the Phy-level hardware based on this configuration value.`

   ```
   NET_PHY_BUS_MODE_MII
   NET_PHY_BUS_MODE_RMII
   NET_PHY_BUS_MODE_SMII
   ```

3. PHY bus type. This field represents the type of electrical attachment of the PHY to the Ethernet controller. In some cases, the PHY may be internal to the network controller, while in other cases, it may be attached via an external MII or RMII bus. It is desirable to specify which attachment method is in use so that a device driver can initialize additional hardware resources if an external PHY is attached to a device that also has an internal PHY. Available settings for this field are:

   ```
   NET_PHY_TYPE_INT
   NET_PHY_TYPE_EXT
   ```

4. Initial PHY link speed. This configuration setting will force the PHY to link to the specified link speed. Optionally, auto-negotiation may be enabled. This field must be set to one of the following values:

   ```
   NET_PHY_SPD_AUTO
    NET_PHY_SPD_10
    NET_PHY_SPD_100
    NET_PHY_SPD_1000
   ```

5. Initial PHY link duplex. This configuration setting will force the PHY to link using the specified duplex. This setting must be set to one of the following values:

   ```
   NET_PHY_DUPLEX_AUTO
   NET_PHY_DUPLEX_HALF
   NET_PHY_DUPLEX_FULL
   ```

Wireless Device Configuration

The listing below shows a sample wireless configuration structure for wireless devices.

```
const  NET_DEV_CFG_WIFI  NetDev_Cfg_WiFi_0 = {
                                           /* Structure member:                           */
    NET_IF_MEM_TYPE_MAIN,                  /*     .RxBufPoolType                          */ (1)
    1518u,                                 /*     .RxBufLargeSize                         */
       9u,                                 /*     .RxBufLargeNbr                          */
      16u,                                 /*     .RxBufAlignOctets                       */
       0u,                                 /*     .RxBufIxOffset                          */

    NET_IF_MEM_TYPE_MAIN,                  /*     .TxBufPoolType                          */
    1606u,                                 /*     .TxBufLargeSize                         */
       4u,                                 /*     .TxBufLargeNbr                          */
     256u,                                 /*     .TxBufSmallSize                         */
       2u,                                 /*     .TxBufSmallNbr                          */
      16u,                                 /*     .TxBufAlignOctets                       */
       0u,                                 /*     .TxBufIxOffset                          */

    0x00000000u,                           /*     .MemAddr                                */
             0u,                           /*     .MemSize                                */

    NET_DEV_CFG_FLAG_NONE,                 /*     .Flag                                   */

    NET_DEV_BAND_DUAL,                     /*     .Band                                    */ (2)

      25000000L,                           /*     .SPI_ClkFreq                             */ (3)
    NET_DEV_SPI_CLK_POL_INACTIVE_HIGH,     /*     .SPI_ClkPol                              */ (4)
    NET_DEV_SPI_CLK_PHASE_FALLING_EDGE,    /*     .SPI_ClkPhase                            */ (5)
    NET_DEV_SPI_XFER_UNIT_LEN_8_BITS,      /*     .SPI_XferUnitLen                         */ (6)
    NET_DEV_SPI_XFER_SHIFT_DIR_FIRST_MSB,  /*     .SPI_XferShiftDir                        */ (7)

   "00:50:C2:25:60:02",                    /*     .HW_AddrStr                              */ (8)
};
```

Listing - Wireless device memory configuration

1. Memory configuration of the wireless device. See µC/TCP-IP Network Interface Configuration for further information about how to configure the memory of your wireless interface.

2. `.Band specifies the desired wireless band enabled and used by the wireless device. This value should be set to one of the following values depending on the hardware capabilities and the application requirements.`

   ```
   NET_DEV_BAND_2_4_GHZ
   NET_DEV_BAND_5_0_GHZ
   NET_DEV_BAND_DUAL
   ```

3. `.SPI_ClkFreq` specifies the SPI controller’s clock frequency (in Hertz) configuration for

writing and reading on the wireless device.

4. `.SPI_ClkPol specifies the SPI controller’s clock polarity configuration for writing and reading on the wireless device. The network device BSP should configure the SPI controller’s clock polarity based on this configuration value.`
   ```
   NET_DEV_SPI_CLK_POL_INACTIVE_LOW
   NET_DEV_SPI_CLK_POL_INACTIVE_HIGH
   ```

5. .SPI_ClkPhase specifies the SPI controller’s clock phase configuration for writing and reading on the wireless device. The network device BSP should configure the SPI controller’s clock phase based on this configuration value.

   ```
   NET_DEV_SPI_CLK_PHASE_FALLING_EDGE
   NET_DEV_SPI_CLK_PHASE_RAISING_EDGE
   ```

6. `.SPI_XferUnitLen specifies the SPI controller’s transfer unit length configuration for writing and reading on the wireless device. The network device BSP should configure the SPI controller’s transfer unit length based on this configuration value.`
   ```
   NET_DEV_SPI_XFER_UNIT_LEN_8_BITS
   NET_DEV_SPI_XFER_UNIT_LEN_16_BITS
   NET_DEV_SPI_XFER_UNIT_LEN_32_BITS
   NET_DEV_SPI_XFER_UNIT_LEN_64_BITS
   ```

7. .SPI_XferShiftDir specifies the SPI controller’s shift direction configuration for writing and reading on the wireless device. The network device BSP should configure the SPI controller’s transfer unit length based on this configuration value.

   ```
   NET_DEV_SPI_XFER_SHIFT_DIR_FIRST_MSB
   NET_DEV_SPI_XFER_SHIFT_DIR_FIRST_LSB
   ```

8. `.HW_AddrStr` specifies the desired device hardware address; may be NULL address or string if the device hardware address is configured or set at run-time. Depending on the driver, if this value is kept NULL or invalid, most device drivers will automatically try to load and use the hardware address located in the memory of the device.

Loopback Configuration

Configuring the loopback interface requires only a memory configuration, as described in µC/TCP-IP Network Interface Configuration.

Listing 5-9 shows a sample configuration structure for the loopback interface.

```
const  NET_IF_CFG_LOOPBACK  NetIF_Cfg_Loopback = {

    NET_IF_MEM_TYPE_MAIN,                                 (1)
    1518,                                                 (2)
      10,                                                 (3)
       4,                                                 (4)
       0,                                                 (5)

    NET_IF_MEM_TYPE_MAIN,                                 (6)
    1594,                                                 (7)
       5,                                                 (8)
     134,                                                 (9)
       5,                                                (10)
       4,                                                (11)
       0,                                                (12)

    0x00000000,                                          (13)
             0,                                          (14)

    NET_DEV_CFG_FLAG_NONE                                (15)
};
```

Listing - Sample loopback interface configuration

1. Receive buffer pool type. This configuration setting controls the memory placement of the receive data buffers. Buffers may either be placed in main memory or in a dedicated, possibly higher speed, memory region (see point #13, below). This field should be set to one of the two macros:

   `NET_IF_MEM_TYPE_MAIN`

   `NET_IF_MEM_TYPE_DEDICATED`

2. Receive buffer size. This field sets the size of the largest receivable packet, and can be set to match the application’s requirements.

   Note: If packets are sent from a socket bound to a non local-host address, to the local host address (127.0.0.1), then the receive buffer size must be configured to match the maximum transmit buffer size, or maximum expected data size, that could be generated from a socket bound to any other interface.

3. Number of receive buffers. This setting controls the number of receive buffers that will be allocated to the loopback interface. This value *must* be set greater than or equal to one buffer if loopback is receiving *only* UDP. If TCP data is expected to be transferred across the loopback interface, then there *must* be a minimum of four receive buffers.

4. Receive buffer alignment. This setting controls the alignment of the receive buffers in bytes. Some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it is probably best practice to align buffers to the data bus width of the processor which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a 4-byte boundary.

5. Receive buffer offset. The loopback interface receives packets starting at base index 0 in the network buffer data areas. This setting configures an offset from the base index of 0 to receive loopback packets. The default offset of 0 *should* be configured. However, if loopback receive packets are configured with an offset, the receive buffer size *must* also be adjusted by the additional number of offset bytes.

6. Transmit buffer pool type. This configuration setting controls the memory placement of the transmit data buffers for the loopback interface. Buffers may either be placed in main memory or in a dedicated, possibly higher speed, memory region (see point #13, below). This field should be set to one of two macros:
   `NET_IF_MEM_TYPE_MAIN`
   `NET_IF_MEM_TYPE_DEDICATED`

7. Large transmit buffer size. At the time of this writing, transmit fragmentation is *not* supported; therefore this field sets the size of the largest transmittable buffer for the loopback interface when the application sends from a socket that is bound to the local-host address.

8. Number of large transmit buffers. This field controls the number of large transmit buffers allocated to the loopback interface. The developer may set this field to zero to make room for additional large transmit buffers, however, the number of large plus the number of small transmit buffers *must* be greater than or equal to one for UDP traffic and three for TCP traffic.

9. Small transmit buffer size. For devices with a minimal amount of RAM, it is possible to allocate small transmit buffers as well as large transmit buffers. In general, we recommend 152 byte small transmit buffers, however, the developer may adjust this value according to the application requirements. This field has no effect if the number of small transmit buffers is configured to zero.

10. Number of small transmit buffers. This field controls the number of small transmit buffers allocated to the device. The developer may set this field to zero to make room for additional large transmit buffers, however, the number of large plus the number of small transmit buffers *must* be greater than or equal to one for UDP traffic and three for TCP traffic.

11. Transmit buffer alignment. This setting controls the alignment of the receive buffers in bytes. Some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it is probably best practice to align buffers to the data bus width of the processor which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a 4-byte boundary.

12. Transmit buffer offset. This setting configures an offset from the base transmit index to prepare loopback packets. The default offset of 0 *should* be configured. However, if loopback transmit packets are configured with an offset, the transmit buffer size *must* also be adjusted by the additional number of offset bytes.

13. Memory address. By default, this field is configured to 0x00000000. A value of 0 tells µC/TCP-IP to allocate buffers for the loopback interface from the µC/LIB Memory Manager default heap. If a faster, more specialized memory is available, the loopback

interface buffers may be allocated into an alternate region if desired.

14. Memory size. By default, this field is configured to 0. A value of 0 tells µC/TCP-IP to allocate as much memory as required from the µC/LIB Memory Manager default heap. If an alternate memory region is specified in the 'Memory Address' field above, then the maximum size of the specified memory segment must be specified.

15. Optional configuration flags. Configure (optional) loopback features by logically **OR** 'ing bit-field flags:
    `NET_DEV_CFG_FLAG_NONE` No loopback configuration flags selected

Adding a Loopback Interface

Basically, to enable and add the loopback interface you only have to enable the loopback interface within the network configuration See Network Interfaces Configuration.

Network Queues Configuration

The device configuration will directly impact the Network Task Queues Configuration.

The µC/TCP-IP stack includes two queues. The first one is the Rx queue and is used to store the Rx buffer that have been filled and are ready to be process. The second queue is the Tx deallocation and is used to store the Tx buffers that are ready to be deallocate.

The size of the Rx queue should reflects the total number of DMA receive descriptors configured for all the interfaces. If the devices are not DMA-based, it should reflects the maximum number of packets that can be acknowledged and signaled during a single receive interrupt even for all interfaces.

The size of the Tx queue should be defined as the total number of small and large transmit buffers declared for all interfaces.

Please refer to section Task Queue Configuration for more details.

LIB Memory Heap Configuration

µC/TCP-IP is using µC/LIB to allocated internal data such as OS objects (semaphore, mutex), device driver's buffers and DMA descriptors, etc. µC/TCP-IP internal tasks stack can be also allocated using µC/LIB. Therefore µC/LIB memory module must be configured properly for µC/TCP-IP. If the heap size is not configured large enough, an error will be returned during the Network Protocol Stack initialization, or during interface addition.

Since the needed heap size is related to the stack configuration (`net_cfg.h`) and is specific to each device driver, it’s not possible to provide an exact formula to calculate it. Thus to optimize the heap size, you should try different heap size until no error is returned for all interfaces added.

Note: The memory module *must* be initialized by the application prior to calling `Net_Init()`. We recommend initializing the memory module before calling starting the *RTOS*, or near the top of the startup task.

Please refer to section µC/LIB Documentation for more details on the µC/LIB module and its configuration.

Heap Memory Calculation for an Interface

The µC/LIB memory heap is used for allocation of the following objects:

1. Transmit small buffers
2. Transmit large buffers
3. Receive large buffers
4. Network Buffers (Network Buffer header and pointer to data area)
5. DMA receive descriptors
6. DMA transmit descriptors
7. Interface data area

8. Device driver data area

9. OS objects (Semaphore, mutex, stack)

10. ICMP Echo request objects (note that object are only allocated when the ICMP Echo request is sent, not at during the Network Protocol Stack initialization)

In the following example, the use of a Network Device Driver with DMA support is assumed. DMA descriptors are included in the analysis. The size of Network Buffer Data Areas (1, 2, 3) vary based on configuration. Refer to Chapter 9, “Buffer Management”. However, for this example, the following object sizes in bytes are assumed:

- Small transmit buffers: 60
- Large transmit buffers: 1518
- Large receive buffers: 1518
- Size of DMA receive descriptor: 8
- Size of DMA transmit descriptor: 8
- Ethernet interface data area: 7
- Average Ethernet device driver data area: 108

With a 4-byte alignment on all memory pool objects, it results in a worst case disposal of three leading bytes for each object. In practice this is not usually true since the size of most objects tend to be even multiples of four. Therefore, the alignment is preserved after having aligned the start of the pool data area. However, this makes the case for allocating objects with size to the next greatest multiple of four in order to prevent lost space due to misalignment.

The approximate memory heap size may be determined according to the following expressions:

```
nbr buf per interface     = nbr small Tx buf +
                            nbr large Tx buf +
                            nbr large Rx buf

nbr net buf per interface = nbr buf per interface

nbr objects   =  nbr buf per interface     +
                 nbr net buf per interface +
                 nbr Rx descriptors        +
                 nbr Tx descriptors        +
                 1 Ethernet      data area +
                 1 Device driver data area

interface mem = (nbr small Tx buf       *   60) +
                (nbr large Tx buf       * 1518) +
                (nbr large Rx buf       * 1518) +
                (nbr Rx descriptors     *    8) +
                (nbr Tx descriptors     *    8) +
                (Ethernet IF  data area *    7) +
                (Ethernet Drv data area *  108) +
                (nbr objects            *    3)

total mem required = nbr interfaces * interface mem
```

Example

With the following configuration, the memory heap required is:

- 10 small transmit buffers
- 10 large transmit buffers
- 10 large receive buffers
- 6 receive descriptors
- 20 transmit descriptors
- Ethernet interface (interface + device driver data area required)

```
nbr     buf per interface = 10 + 10 + 10               = 30
          nbr net buf per interface = nbr buf per interface      = 30
          nbr objects               = (30 + 30 + 6 + 20 + 1 + 1) = 88
          interface mem             = (10 *   60) +
                                      (10 * 1518) +
                                      (10 * 1518) +
                                      ( 6 *    8) +
                                      (20 *    8) +
                                      ( 1 *    7) +
                                      ( 1 *  108) +
                                      (88 *    3) = 31,547 bytes

          total mem required > 31,547 ( + localhost memory, if enabled)
```

The localhost interface, when enabled, requires a similar amount of memory except that it does not require Rx and Tx descriptors, an IF data area, or a device driver data area.

The value determined by these expressions is only an estimate. In some cases, it may be possible to reduce the size of the μC/LIB memory heap by inspecting μC/LIB (see μC/LIB Documentation) after all interfaces have been successfully initialized and any additional application allocations (if applicable) have been completed.

TCPIP Initialization Guide

This section describes the different steps to initialize the μC/TCP-IP Stack.

The last section of this guide also provides examples of TCP/IP stack initialization.

- Prerequisite module initialization
- Initializing Tasks and objects
- Initializing Interfaces
- IP Address Configuration
- Initializing+Shell+commands
- Sample applications

Prerequisite module initialization

Before initializing µC/TCPIP some prerequisite modules must be initialized prior to starting the Network Protocol stacks initialization. µC/TCPIP requires an RTOS such as µCOS-II or µCOS-III. Before starting initializing µC/TCPIP and other prerequisite modules, the RTOS must be started and all initialization call should be performed within an initialization task. Please refer to the user manual of your RTOS for more information about how to initialize the RTOS and how to initialize other modules.

µC/TCPIP is using µC/CPU, µC/LIB memory module and µC/Common Kernel Abstraction Layer, refer to the following documentation for more information about the initialization of these modules:

µC/LIB User's Guide

µC/CPU User's Manual

µC/Common Documentation

Listing - AppTaskStart shows an example of the application initialization task that should be started by the RTOS. The listing shows also what prerequisite modules that must absolutely initialized prior calling the TCP/IP function `AppInit_TCPIP(),` which would be responsible to initialize the Network protocol tacks. Section Sample applications will detailed the contain of the AppInit_TCPIP() function depending on the type of interface used.

```
static  void  AppTaskStart (void *p_arg)
{
    CPU_INT32U  cpu_clk_freq;
    CPU_INT32U  cnts;
    OS_ERR      err_os;
    KAL_ERR     kal_err;
    NET_ERR     net_err;

    (void)&p_arg;

    BSP_Init();                                                 (1)
    CPU_Init();
    Mem_Init();

    AppInit_TCPIP(&net_err);                                    (2)
                                                                (3)

    while (1) {
        OSTimeDlyHMSM((CPU_INT16U) 0u,
                      (CPU_INT16U) 0u,
                      (CPU_INT16U) 0u,
                      (CPU_INT16U) 100u,
                      (OS_OPT    ) OS_OPT_TIME_HMSM_STRICT,
                      (OS_ERR   *)&err_os);
    }
}
```

Listing - AppTaskStart

(1) `BSP_Init()`, `CPU_Init()` and `Mem_Init()` must be called prior to the TCP-IP initialization function `AppInit_TCPIP()`.

(2) `AppInit_TCPIP()` initializes the μC/TCP-IP stack and the initial parameters to configure it.

(3) If other IP applications are required this is where they are initialized.

Initializing Tasks and objects

After all the µC/TCP-IP prerequisite modules have been initialized (see section Prerequisite Module Initialization), the TCP/IP stack must be initialize with the function `Net_Init()`. This function must be called before any other network API functions.

This function will create the OS objects required by the TCP/IP module, initialize to their default value all the network configurable parameters, initialize the network statistic counters, initialize the network buffer pools, initialize all the different network layers, etc.

This function also takes as arguments the three Network Task configurations defined in the `net_cfg.c` (see section Network Stack Configuration) file as shown in the function prototype below.

```
NET_ERR Net_Init(NET_TASK_CFG  *p_rx_task_cfg,
                 NET_TASK_CFG  *p_tx_task_cfg,
                 NET_TASK_CFG  *p_tmr_task_cfg);
```

For more details on the `Net_Init()` function refer to API functions section here.

The section Sample Applications also gives examples of TCP/IP application initialization functions.

Initializing Interfaces

Initialize an Interface

Adding an Interface

Interfaces may be added to the stack by calling `NetIF_Add()`. Each new interface requires additional BSP. The order of addition is critical to ensure that the interface number assigned to the new interface matches the code defined within `net_bsp.c`. See section Network Interface Configuration for more information on configuring interfaces.

Starting an Interface

Interfaces may be started by calling `NetIF_Start()`. See section Starting and Stopping Network Interfaces for more information on starting interfaces.

Initialize an Ethernet Interface

Once µC/TCP-IP is initialized, each network interface must be added to the stack via `NetIF_Add()` function. `NetIF_Add()` validates the network interface arguments, initializes the interface, and adds it to the interface list of the TCP/IP stack. µC/TCP-IP uses a structure that contains pointers to API functions which are used to access the interface layer, and configuration structures are used to initialize resources needed by the network interface. You must pass the following arguments to the `NetIF_Add()` function:

```
NET_IF_NBR  NetIF_Add (void     *if_api,                              (1)
                       void     *dev_api,                             (2)
                       void     *dev_bsp,                             (3)
                       void     *dev_cfg,                             (4)
                       void     *ext_api,                             (5)
                       void     *ext_cfg,                             (6)
                       NET_ERR  *perr)                                (7)
```

Listing - NetIF_Add() arguments

1. The first argument specifies the link layer API pointers structure that will receive data from the hardware device. For an Ethernet interface, this value will always be defined as `NetIF_API_Ether`. This symbol is defined by µC/TCP-IP and it can be used to add as many Ethernet network interface’s as necessary. This API should always be provided

with the TCP-IP stack which can be found under the interface folder (`/IF/net_if_ether.*`).

2. The second argument represents the hardware device driver API pointers structure which is defined as a fixed structure of function pointers of the type specified by Micriμm for use with µC/TCP-IP. If Micriμm supplies the device driver, the symbol name of the device API will be defined within the device driver at the top of the device driver source code file. You can find the device driver under the device folder (`/Dev/Ether/<controller>`). Otherwise, the driver developer is responsible for creating the device driver and the API structure should start from the device driver template which can be found under the device folder (`/Dev/Ether/Template`).

3. The third argument specifies the specific device's board-specific (BSP) interface functions which is defined as a fixed structure of function pointers. The application developer must define both the BSP interface structure of function pointers and the actual BSP functions referenced by the BSP interface structure and should start from the BSP template provided with the stack which you can find under the BSP folder (`/BSP/Template`). Micriμm may be able to supply example BSP interface structures and functions for certain evaluation boards. For more information about declaring BSP interface structures and BSP functions device, see section Network Board Support Package for further information about the BSP API.

4. The fourth argument specifies the device driver configuration structure that will be used to configure the device hardware for the interface being added. The device configuration structure format has been specified by Micriμm and must be provided by the application developer since it is specific to the selection of device hardware and design of the evaluation board. Micriμm may be able to supply example device configuration structures for certain evaluation boards. For more information about declaring a device configuration structure, see section Ethernet Device Configuration section.

5. The fifth argument represents the physical layer hardware device API. In most cases, when Ethernet is the link layer API specified in the first argument, the physical layer API may be defined as `NetPHY_API_Generic`. This symbol has been defined by the generic Ethernet physical layer device driver which can be supplied by Micriμm. If a

custom physical layer device driver is required, then the developer would be responsible for creating the API structure. Often Ethernet devices have built-in physical layer devices which are *not* (R)MII compliant. In this circumstance, the physical layer device driver API field may be left NULL and the Ethernet device driver may implement routines for the built-in PHY.

6. The sixth argument represents the physical layer hardware device configuration structure. This structure is specified by the application developer and contains such information as the physical device connection type, address, and desired link state upon initialization. For devices with built in non (R)MII compliant physical layer devices, this field may be left `NULL`. However, it may be convenient to declare a physical layer device configuration structure and use some of the members for physical layer device initialization from within the Ethernet device driver. For more information about declaring a physical layer hardware configuration structure, see section Ethernet PHY Configuration.

7. The last argument is a pointer to a `NET_ERR` variable that contains the return error code for `NetIF_Add()`. This variable should be checked by the application to ensure that no errors have occurred during network interface addition. Upon success, the return error code will be `NET_IF_ERR_NONE`.

Note: If an error occurs during the call to `NetIF_Add()`, the application *may* attempt to call `NetIF_Add()` a second time for the same interface but unless a temporary hardware fault occured, the application developer should observe the error code, determine and resolve the cause of the error, rebuild the application and try again. If a hardware failure occurred, the application may attempt to add an interface as many times as necessary, but a common problem to watch for is a µC/LIB Memory Manager heap out-of-memory condition. This may occur when adding network interfaces if there is insufficient memory to complete the operation. If this error occurs, the configured size of the µC/LIB heap within `lib_cfg.h` must be increased.

Once an interface is added successfully, the next step is to configure the interface with one or more network layer protocol addresses.

For a thorough description of the µC/TCP-IP files and directory structure, see section Directories and Files.

When the network interface is added without error, it must be started via the `NetIF_Start()` function to be available and be used by the µC/TCP-IP. The following code example shows how to initialize µC/TCP-IP, add an interface, configure the IP address and start it:

```
#include  <Source/net.h>
#include  <net_dev_dev1.h>
#include  <net_bsp.h>
#include  <net_phy.h>
CPU_BOOLEAN  App_InitTCPIP (void)
{
    NET_IF_NBR     if_nbr;
    NET_ERR        err;

    err = Net_Init(p_rx_task_cfg,
                   p_tx_task_cfg,
                   p_tmr_task_cfg);
    if (err != NET_ERR_NONE) {
        return (DEF_FAIL);
    }
    if_nbr  = NetIF_Add((void    *)&NetIF_API_Ether
                        (void    *)&NetDev_API_Etherxxx,
                        (void    *)&NetDev_BSP_API,
                        (void    *)&NetDev_Cfg_Ether_0,
                        (void    *)&NetPhy_API_Generic,
                        (void    *)&NetPhy_Cfg_0,
                        (NET_ERR *)&err);
    if (err != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }

    NetIF_Start(if_nbr, &err);
    if (err != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }

 return (DEF_OK);
}
```

Listing - Ethernet interface initialization example

Initialize an Wireless Interface

Once µC/TCP-IP is initialized each network interface must be added to the stack via the `NetIF_Add()` function which validates the network interface arguments, initializes the interface and adds it to the interface list. µC/TCP-IP uses a structure that contains pointers to API functions which are used to access the interface layer and configuration structures are used to initialize resources needed by the network interface. You must pass the following arguments to the `NetIF_Add()` function:

```
NET_IF_NBR  NetIF_Add (void     *if_api,                        (1)
                       void     *dev_api,                       (2)
                       void     *dev_bsp,                       (3)
                       void     *dev_cfg,                       (4)
                       void     *ext_api,                       (5)
                       void     *ext_cfg,                       (6)
                       NET_ERR  *perr)                          (7)
```

Listing - NetIF_Add() arguments

1. The first argument specifies the link layer API pointers structure that will receive data from the hardware device. For a wireless interface, this value will always be defined as `NetIF_API_WiFi`. This symbol is defined by µC/TCP-IP and it can be used to add as many wireless network interfaces as necessary. This API should always be provided with the TCP-IP stack which can be found under the interface folder (`/IF/net_if_wifi.*`).

2. The second argument represents the hardware device driver API which is defined as a fixed structure of function pointers of the type specified by Micriµm for use with µC/TCP-IP. If Micriµm supplies the device driver, the symbol name of the device API will be defined within the device driver at the top of the device driver source code file. You can find the device driver under the device folder (`/Dev/WiFi/<device>`). Otherwise, the driver developer is responsible for creating the device driver and the API structure should start from the device driver template which can be found under the device folder (`/Dev/WiFi/Template`).

3. The third argument specifies the specific device's board-specific (BSP) interface functions which are defined as a fixed structure of function pointers. The application developer must define both the BSP interface structure of function pointers and the actual BSP functions referenced by the BSP interface structure and should start from the BSP template provided with the stack which you can find under the BSP folder (`/BSP/Template`). Micrium may be able to supply example BSP interface structures and functions for certain evaluation boards. For more information about declaring BSP interface structures and the BSP functions device, see Network Board Support Package for further information about the BSP API.

4. The fourth argument specifies the device driver configuration structure that will be used

to configure the device hardware for the interface being added. The device configuration structure format has been specified by Micriµm and must be provided by the application developer since it is specific to the selection of device hardware and design of the evaluation board. Micriµm may be able to supply example device configuration structures for certain evaluation boards. For more information about declaring a device configuration structure, see Wireless Device Configuration.

5. The fifth argument represents the extension layer device API. In most cases, when wireless is the Wireless Manager layer API specified in the first argument, the Wireless Manager layer API may be defined as `NetWiFiMgr_API_Generic`. This symbol has been defined by the generic Wireless Manager layer which can be supplied by Micriµm. If a custom Wireless Manager layer is required, then the developer would be responsible for creating the API structure.

6. The sixth argument represents the extension layer configuration structure. This structure is specified by the application developer. For devices which use the generic Wireless Manager this field should be left `NULL`. However, it may be convenient to declare a Wireless Manager layer device configuration structure and use some of the members for Wireless Manager layer initialization from within the wireless device driver or a custom Wireless Manager.

7. The last argument is a pointer to a `NET_ERR` variable that contains the return error code for `NetIF_Add()`. This variable *should* be checked by the application to ensure that no errors have occurred during network interface addition. Upon success, the return error code will be `NET_IF_ERR_NONE`.

Note: If an error occurs during the call to `NetIF_Add()`, the application *may* attempt to call `NetIF_Add()` a second time for the same interface but unless a temporary hardware fault occurred, the application developer should observe the error code, determine and resolve the cause of the error, rebuild the application and try again. If a hardware failure occurred, the application may attempt to add an interface as many times as necessary, but a common problem to watch for is a µC/LIB Memory Manager heap out-of-memory condition. This may occur when adding network interfaces if there is insufficient memory to complete the operation. If this error occurs, the configured size of the µC/LIB heap within `app_cfg.h` must be increased.

Once an interface is added successfully, the next step is to configure the interface with one or more network layer protocol addresses.

For a thorough description of the µC/TCP-IP files and directory structure, see Directories and Files.

Once a network interface is added without error, it must be started via the `NetIF_Start()` function to be seen as available and to be used by µC/TCP-IP. The following code example shows how to initialize µC/TCP-IP, add an interface, add an IP address and start the interface:

```
#include  <net.h>
#include  <net_dev_rs9110n2x.h>
#include  <net_bsp.h>
#include  <net_phy.h>
CPU_BOOLEAN  App_InitTCPIP (void)
{
    NET_IF_NBR   if_nbr;
    NET_ERR      err;

    err = Net_Init();
    if (err != NET_ERR_NONE) {
        return (DEF_FAIL);
    }
    if_nbr  = NetIF_Add((void    *)&NetIF_API_WiFi
                        (void    *)&NetDev_API_RS9110N2x,
                        (void    *)&NetDev_BSP_SPI_API,
                        (void    *)&NetDev_Cfg_WiFi_0,
                        (void    *)&NetWiFiMgr_API_Generic,
                        (void    *) 0,
                        (NET_ERR *)&err);
    if (err != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }

    NetIF_Start(if_nbr, &err);
    if (err != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }

 return (DEF_OK);
}
```

Listing - Wireless interface initialization example

IP Address Configuration

The following sections provide sample code describing how to configure IP address (IPv4 and IPv6).

For a complete guide on IP addressing, refer to section IP Address Programming.

Configuring an IP Address on an Interface

Each network interface must be configured with at least one IP address. It could be an IPv4 or an IPv6 address or both depending on which modules the TCP-IP stack has enabled.

IPv4

For IPv4, the address configuration may be performed using µC/DHCPc or manually during run-time. If run-time configuration is chosen, the following functions may be utilized to set the IPv4, network mask, and gateway addresses for a specific interface.

`NetASCII_Str_to_IP`

`NetIPv4_CfgAddrAdd`

More than one set of IPv4 addresses may be configured for a specific network interface by calling the functions above. The constant `NET_IPv4_CFG_IF_MAX_NBR_ADDR` specified in `net_cfg.h` determines the maximum number of IPv4 addresses that may be assigned to an interface.

Note that on the default interface, the first IPv4 address added will be the default address used for all default IPv4 communication.

The first function aids the developer by converting a string format IPv4 address such as “192.168.1.2” to its hexadecimal equivalent. The second function is used to configure an interface with the specified IPv4, network mask and gateway addresses. An example is shown in listing Listing - IPv4 Address Configuration Example.

```
CPU_BOOLEAN    cfg_success;
NET_IPv4_ADDR  ipv4_addr;
NET_IPv4_ADDR  ipv4_msk;
NET_IPv4_ADDR  ipv4_gateway;
NET_ERR        err;

(void)NetASCII_Str_to_IP((CPU_CHAR*)”192.168.1.2”,   &ipv4_addr,    NET_IPv4_ADDR_SIZE, &err); /* See
Note #1 */
(void)NetASCII_Str_to_IP((CPU_CHAR*)”255.255.255.0”, &ipv4_msk,     NET_IPv4_ADDR_SIZE, &err);
(void)NetASCII_Str_to_IP((CPU_CHAR*)”192.168.1.1”,   &ipv4_gateway, NET_IPv4_ADDR_SIZE, &err);

cfg_success = NetIPv4_CfgAddrAdd(if_nbr,                    /* See Note #2 */
                                 ipv4_addr,                 /* See Note #3 */
                                 ipv4_msk,                  /* See Note #4 */
                                 ipv4_gateway,              /* See Note #5 */
                                &err);                      /* See Note #6 */
```

Listing - IPv4 Address Configuration Example

1. `NetASCII_Str_to_IP()` requires four arguments. The first function argument is a string representing a valid IP address. The second argument is a pointer to the IP address object that will received the conversion result. The third argument is the size of the address object and the last argument is a pointer to a `NET_ERR` to contain the return error code. Upon successful conversion, the return error will contain the value `NET_ASCII_ERR_NONE` and the function will return a variable of type `NET_IP_ADDR_FAMILY` containing the family type (IPv4 or IPv6) of the address converted.

2. The first argument is the number representing the network interface that is to be configured. This value is obtained as the result of a successful call to `NetIF_Add()`.

3. The second argument is the `NET_IPv4_ADDR` value representing the IPv4 address to be configured.

4. The third argument is the `NET_IPv4_ADDR` value representing the subnet mask address that is to be configured.

5. The fourth argument is the `NET_IPv4_ADDR` value representing the default gateway IPv4 address that is to be configured.

6. The fifth argument is a pointer to a `NET_ERR` variable containing the return error code for the function. If the interface address information is configured successfully, then the return error code will contain the value `NET_IPv4_ERR_NONE`. Additionally, function

returns a Boolean value of `DEF_OK` or `DEF_FAIL` depending on the result. Either the return value or the `NET_ERR` variable may be checked for return status; however, the `NET_ERR` contains more detailed information and should therefore be the preferred check.

IPv6

Currently, the µC/TCP-IP stack only support manual static IPv6 address configuration and IPv6 Stateless Address Auto-Configuration. Dynamic address configuration with DHCPv6 is not yet supported.

Manual Static Address Configuration

the following functions may be utilized to set the IPv6 address for a specific interface:

```
NetASCII_Str_to_IP
NetIPv6_CfgAddrAdd
NetIPv6_CfgAddrHookSet
```

More than one set of IPv6 addresses may be configured for a specific network interface by calling the functions above. The constant `NET_IPv6_CFG_IF_MAX_NBR_ADDR` specified in `net_cfg.h` determines the maximum number of IPv6 addresses that may be assigned to an interface.

Note that on the default interface, the first IPv6 address added will be the default address used for all default IPv6 communication.

The first function aids the developer by converting a string format IPv6 address such as “fe80::1111:1111” to its network equivalent. The second function is used to configure an interface with the specified IPv6 address. An example is shown in listing Listing - IPv6 Address Configuration Example.

```
CPU_BOOLEAN    cfg_success;
NET_IPv6_ADDR  ipv6_addr;
NET_FLAGS      ipv6_flags;
NET_ERR        err;


(void)NetASCII_Str_to_IP((CPU_CHAR *)”fe80::1111:1111”,  /* See Note #1 */
                                     &ipv6_addr,
                                      NET_IPv6_ADDR_SIZE,
                                     &err);

ipv6_flags = 0;
DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_BLOCK_EN);            /* See Note #2 */
DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_DAD_EN);              /* See Note #3 */

cfg_success = NetIPv6_CfgAddrAdd(if_nbr,            /* See Note #4 */
                                &ipv6_addr,         /* See Note #5 */
                                 64,                /* See Note #6 */
                                 ipv6_flags,                /* See Note #7 */
                                &err);              /* See Note #8 */
```

Listing - IPv6 Address Configuration Example

1. See NetASCII_Str_to_IP for more details.

2. Set Address Configuration as blocking.

3. Enable DAD with Address Configuration.

4. The first argument is the number representing the network interface that is to be configured. This value is obtained as the result of a successful call to `NetIF_Add()`.

5. The second argument is the pointer to the `NET_IPv6_ADDR` value representing the IPv6 address to be configured.

6. The third argument is the IPv6 prefix length of the addresss to configured.

7. The fourth argument is a set of network flags holding options specific to the address configuration process.

8. The fifth argument is a pointer to a `NET_ERR` variable containing the return error code for the function. If the interface address information is configured successfully, then the return error code will contain the value `NET_IPv6_ERR_NONE`. Additionally, function returns a Boolean value of `DEF_OK` or `DEF_FAIL` depending on the result. Either the return value or the `NET_ERR` variable may be checked for return status; however, the `NET_ERR` contains more detailed information and should therefore be the preferred check.

As shown in Listing - IPv6 Address Configuration Example, the NetIPv6_CfgAddrAdd() function can take as argument a set of network flags. The following options are available :

Flags	Description
`NET_IPv6_FLAG_BLOCK_EN`	Enables blocking mode.
`NET_IPv6_FLAG_DAD_EN`	Enables Duplication Address Detection (DAD) with the address configuration process.

It is therefore possible to make the function blocking or not, or to enable Duplication Address Detection with the address configuration.

If the function is made none blocking, it is possible to set a hook function to advertise the application when the address configuration process has finished. The API function NetIPv6_CfgAddrHookSet can be used to set the hook function. Refer to section IPv6 Static Address Configuration Hook Function for all the details on the hook function format and usage. Listing - Non-Blocking IPv6 Address Configuration Example in the *IP Address Configuration* page shows an example of a non-blocking IPv6 static address configuration.

```
CPU_BOOLEAN    cfg_success;
NET_IPv6_ADDR  ipv6_addr;
NET_FLAGS      ipv6_flags;
NET_ERR        err;


(void)NetASCII_Str_to_IP((CPU_CHAR *)"fe80::1111:1111",  /* Convert IPv6 string address to 128 bit 
address.      */
                                    &ipv6_addr,
                                     NET_IPv6_ADDR_SIZE,
                                    &err);

NetIPv6_SetAddrCfgHookFnct(if_nbr,                          /* Set hook function to received addr cfg 
result.       */
                          &App_AddrCfgResult,    /* TODO update pointer to hook fnct implemented in App. 
*/
                          &err_net);

ipv6_flags = 0;
DEF_BIT_CLR(ipv6_flags, NET_IPv6_FLAG_BLOCK_EN);            /* Set Address Configuration as 
non-blocking.           */
DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_DAD_EN);              /* Enable DAD with Address Configuration.  
*/

cfg_success = NetIPv6_CfgAddrAdd(if_nbr,            /* Add a statically-configured IPv6 host address to 
... */
                                &ipv6_addr,         /* ... the interface.                            
*/
                                 64,
                                 ipv6_flags,
                                &err);
```

Listing - Non-Blocking IPv6 Address Configuration Example

Stateless Address Auto-Configuration

The IPv6 protocol defines an address Auto-Configuration procedure allowing a network interface to set itself an IPv6 Link-Local address based on its Interface ID. The Auto-Configuration process will also query the local network to found an IPv6 router that could send prefix information to set an IPv6 global address.

The µC/TCP-IP stack supports only the EUI-64 format for interface ID. This format creates a 64 bits ID based on the 48 bits MAC address of the interface. Those 64 bits will become the 64 least significant bits of the IPv6 addresses configured with the Stateless Auto-Configuration process.

The following functions may be used to configure the IPv6 Stateless Auto-Configuration process:

NetIPv6_AddrAutoCfgEn

NetIPv6_AddrAutoCfgDis

NetIPv6_AddrAutoCfgHookSet

The IPv6 Auto-Configuration procedure inside the µC/TCP-IP stack is a non-blocking process. To recover the result of the Auto-Configuration, a hook function can be configured that will be called by the TCP/IP stack when the Auto-Configuration has finished. The API function used to set the hook function is NetIPv6_AddrAutoCfgHookSet. Refer to section IPv6 Stateless Address Auto-Configuration Hook Function for all the details on the Auto-Configuration hook function format and usage and refer to section Sample applications for examples of Auto-Configuration.

Initializing+Shell+commands

The command line interface is a traditional method for accessing the network functionality on a remote system (telnet), or in a device with a serial port (be that RS-232 or USB). A group of shell commands are available for μC/TCPIP. These may simply expedite evaluation of the network suite, or become part a primary method of access (or gathering debug information) in your final product.

Using the Shell Commands

To use shell commands some file, in addition to the generic μC/TCPIP files, must be included in the build:

See Directories and Files - Shell Commands

Using Network Interface Programming API

The following files must be included in any application or header files initialize μC/Shell or handle shell commands.

Include file	Description
Cmd/net_cmd.h	Contains the initialization function API

API Reference

All Interface APIs are presented in the section

Function Name	Description
NetCmd_Init()	Add Network Shell commands to μC/Shell

Initialization order

Modules must be initialized in the following order:

1. μC/Shell

2. Network suite (see Initializing Tasks and objects)

3. Network Shell Command

```
#include  <Source/net.h>
#include  <net_dev_rs9110n2x.h>
#include  <net_bsp.h>
#include  <net_phy.h>
#include  <Cmd/net_cmd.h>
CPU_BOOLEAN  App_InitTCPIP (void)
{
    NET_IF_NBR   if_nbr;
    NET_ERR      err;
    NET_CMD_ERR  err_cmd;

    err = Net_Init();
    if (err != NET_ERR_NONE) {
        return (DEF_FAIL);
    }
    if_nbr  = NetIF_Add((void    *)&NetIF_API_WiFi
                        (void    *)&NetDev_API_RS9110N2x,
                        (void    *)&NetDev_BSP_SPI_API,
                        (void    *)&NetDev_Cfg_WiFi_0,
                        (void    *)&NetWiFiMgr_API_Generic,
                        (void    *) 0,
                        (NET_ERR *)&err);
    if (err != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }

    NetIF_Start(if_nbr, &err);
    if (err != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }

    /*uC-Shell must has been initialized before initializing Network Shell Command */
    NetCmd_Init(&err_cmd);
    if (err != NET_CMD_ERR_NONE) {
        return (DEF_FAIL);
    }

    return (DEF_OK);
}
```

Sample applications

This section presents distinct examples of TCP/IP application initialization functions for the different type of network Interface (Ethernet and WiFi):

- Ethernet Sample Application
- WiFi Sample Application
- Multiple Interfaces Sample Application

Ethernet Sample Application

1. This example show how to initialize µC/TCP-IP:

    a. Initialize Stack tasks and objects

    b. Initialize an Ethernet Interface

    c. Start that Ethernet Interface

    d. Configure IP addresses of that Interface

This example is based on template files so some modifications will be required, insert the appropriate project/board specific code to perform the stated actions. Note that the file init_ether.c, located in the folder `$/Micrium/Software/uC-TCPIP/Examples/Init`, contains this sample application:

```
#include  <cpu_core.h>
#include  <lib_mem.h>
#include  <Source/net.h>
#include  <Source/net_ascii.h>
#include  <IF/net_if.h>
#include  <IF/net_if_ether.h>
#ifdef NET_IPv4_MODULE_EN
#include  <IP/IPv4/net_ipv4.h>
#endif
#ifdef NET_IPv6_MODULE_EN
#include  <IP/IPv6/net_ipv6.h>
#endif
#include  <Cfg/Template/net_dev_cfg.h>                      /* See Note #1. */
#include  <Dev/Ether/Template/net_dev_ether_template_dma.h> /* See Note #2. */
#include  <Dev/Ether/PHY/Generic/net_phy.h>                 /* See Note #3. */
#include  <BSP/Template/net_bsp_ether.h>                    /* See Note #4. */


CPU_BOOLEAN  AppInit_TCPIP (void)
{
    NET_IF_NBR      if_nbr;
    NET_ERR         err_net;
#ifdef NET_IPv4_MODULE_EN
    NET_IPv4_ADDR   addr_ipv4;
    NET_IPv4_ADDR   msk_ipv4;
    NET_IPv4_ADDR   gateway_ipv4;
#endif
#ifdef NET_IPv6_MODULE_EN
    CPU_BOOLEAN     cfg_result;
#if (NET_IPv6_CFG_ADDR_AUTO_CFG_EN == DEF_DISABLED)
    NET_FLAGS       ipv6_flags;
    NET_IPv6_ADDR   ipv6_addr;
#endif
#endif
                                                        /* ------- PREREQUISITES MODULE INIT -------- */
    CPU_Init();                                         /* See Note #5.                               */
    Mem_Init();
                                                        /* ------ INIT NETWORK TASKS & OBJECTS ------ */
    err_net = Net_Init(&NetRxTaskCfg,                   /* See Note #6.                               */
                       &NetTxDeallocTaskCfg,
                       &NetTmrTaskCfg);
    if (err_net != NET_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                        /* --------- ADD ETHERNET INTERFACE --------- */
                                                        /* See Note #7.                               */
    if_nbr = NetIF_Add(&NetIF_API_Ether,                /* See Note #7b.                              */
                       &NetDev_API_TemplateEtherDMA,    /* Device driver API,    See Note #7c.        */
                       &NetDev_BSP_BoardDev_Nbr,        /* BSP API,              See Note #7d.        */
                       &NetDev_Cfg_Ether_1,             /* Device configuration, See Note #7e.        */
                       &NetPhy_API_Generic,             /* PHY driver API,       See Note #7f.        */
                       &NetPhy_Cfg_Ether_1,             /* PHY configuration,    See Note #7g.        */
                       &err_net);
    if (err_net != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                        /* -------- START ETHERNET INTERFACE -------- */
    NetIF_Start(if_nbr, &err_net);                      /* See Note #8.                               */
    if (err_net != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }
#ifdef NET_IPv4_MODULE_EN
                                                        /* ------- CONFIGURE IPV4 STATIC ADDR ------- */
                                                        /* See Note #9                                */
    NetASCII_Str_to_IP("10.10.10.64",                   /* Convert IPv4 string addr to 32 bits addr.  */
                       &addr_ipv4,
                       NET_IPv4_ADDR_SIZE,
                      &err_net);
    NetASCII_Str_to_IP("255.255.255.0",                 /* Convert IPv4 mask string to 32 bits addr.  */
                       &msk_ipv4,
                       NET_IPv4_ADDR_SIZE,
                      &err_net);
    NetASCII_Str_to_IP("10.10.10.1",                    /* Convert Gateway string to 32 bits addr.    */
                       &gateway_ipv4,
                       NET_IPv4_ADDR_SIZE,
                      &err_net);
    NetIPv4_CfgAddrAdd(if_nbr,                          /* Add a statically-configured IPv4 host ...  */
                       addr_ipv4,                       /* ... addr, subnet mask, & default      ...  */
                       msk_ipv4,                        /* ... gateway to the interface. See Note #10.*/
                       gateway_ipv4,
                      &err_net);
    if (err_net != NET_IPv4_ERR_NONE) {
        return (DEF_FAIL);
    }
#endif
#ifdef NET_IPv6_MODULE_EN
#if (NET_IPv6_CFG_ADDR_AUTO_CFG_EN == DEF_ENABLED)
                                                        /* ---- START IPV6 STATELESS AUTO-CONFI ----- */
                                                        /* See Note #12.                              */
    NetIPv6_AddrAutoCfgHookSet(if_nbr,                  /* Set hook to received Auto-Cfg result.      */
                              &App_AutoCfgResult,       /* TODO update pointer to hook defined in App.*/
                              &err_net);

                                                        /* See Note #13.                              */
    cfg_result = NetIPv6_AddrAutoCfgEn(if_nbr,          /* Enable and Start Auto-Config process.      */
                                       DEF_YES,
                                      &err_net);
    if (cfg_result == DEF_FAIL) {
        return (DEF_FAIL);
    }
```

Listing - AppInit_TCPIP()

1. The device configuration template file should be copied to your application folder and modified to follow your requirements. Refer to the User's Manual for more information about how to configure your device.

   We recommend starting with a working configuration from an example project for your MCU. Micriµm might have some projects available only for internal usage, so if no working project are found online, please ask at support@micrium.com for a configuration file example.

2. Most of the time Micriµm provides an Ethernet Network device driver which can be found under the following folder:

   ```
   $/Micrium/Software/uC-TCPIP/Dev/Ether/<Controller>/net_dev_<controller>.h
   ```

   If Micrium does not support your network device driver, you will have to write your own device driver starting from the Ethernet Device driver template. Before starting to write your own driver, make sure that the driver is not already available. `Net_Init()` is the Network Protocol stack initialization function. This function takes the three TCP-IP internal tasks configurations (defined in `net_cfg.c`) as argument.

3. The PHY driver should be provided by Micriµm and located under the following folder:

```
$/Micrium/Software/uC-TCPIP/Dev/Ether/PHY/<phy part number>/net_phy_<phy part
number>.h
```

   Most of the time for MII, RMII or GMII PHY, the generic PHY works correctly. If your PHY is not available and the generic is not working you will have to write you own PHY driver. Normally for a single connector PHY, some minor changes to the generic driver are required.

4. The board support package (BSP) template file should be copied to your application folder and modified for your specific board. Refer to the User's Manual for more information about how to write a BSP (Ethernet BSP Layer).

   However we recommend, starting with a working configuration from an example project for your MCU. Micriµm might have some projects available only for internal usage, so if no working project are found online, please ask at support@micrium.com for a BSP file example specific for your MCU.

5. Some prerequisite module initializations are required. The following modules must be initialized prior to starting the Network Protocol stacks initialization:

   a. uC/CPU

   b. uC/LIB Memory module

6. `Net_Init()` is the Network Protocol stack's initialization function. It must only be called once and before any other Network functions.

   a. This function takes the three TCP-IP internal task configuration structures as arguments (such as priority, stack size, etc.). By default these configuration structures are defined in `net_cfg.c` :

```
NetRxTaskCfg RX task configuration
NetTxDeallocTaskCfg TX task configuration
NetTmrTaskCfg Timer task configuration
```

b. We recommend you configure the Network Protocol Stack task priorities & Network application (such as a Web server) task priorities as follows:

```
NetTxDeallocTaskCfg                             (highest priority)

Network applications (HTTPs, FTP, DNS, etc.)

NetTmrTaskCfg
NetRxTaskCfg                                    (lowest priority)
```

We recommend that the uC/TCP-IP Timer task and network interface Receive task be lower priority than almost all other application tasks; but we recommend that the network interface Transmit De-allocation task be higher priority than all application tasks that use uC/TCP-IP network services.

However, better performance can be observed when the Network applications are set with the lowest priority. Some experimentation could be required to identify the best task priority configuration.

7. `NetIF_Add()` is a network interface function responsible for initializing a Network Device driver.

a. `NetIF_Add()` returns the interface index number. The interface index number should start at '1', since the interface '0' is reserved for the loopback interface. The interface index number must be used when you want to access the interface using any Network interface API.

b. The first parameter is the address of the Network interface API. These API are provided by Micriµm and are defined in file 'net_if_<type>.h'. It should be either:

```
NetIF_API_Ether     Ethernet interface
NetIF_API_WiFi      Wireless interface
```

c. The second parameter is the address of the device API function. The API should be defined in the Device driver header:

```
$/uC-TCPIP/Dev/<if_type>/<controller>/net_dev_<controller>.h
```

d. The third parameter is the address of the device BSP data structure. See 'Note #4' for more details.

e. The fourth parameter is the address of the device configuration data structure. See 'Note #1' for more details.

f. The fifth parameter is the address of the PHY API function. See Note #3' for more details.

g. The sixth and last parameter is the address of the PHY configuration data structure. The PHY configuration should be located in net_dev_cfg.c/h.

8. `NetIF_Start()` makes the network interface ready to receive and transmit. Once this function returns without an error the device should be able to receive packet, an interrupt should then be generated from the Ethernet controller (at least for each packets present on the cable).

9. `NetASCII_Str_to_IP()` converts the human readable address into a format required by the protocol stack.

   In this example the IP address used, 10.10.10.64, addresses out of the 10.10.10.1 network with a subnet mask of 255.255.255.0. To match different networks, the IP address, the subnet mask and the default gateway's IP address have to be customized.

10. `NetIPv4_CfgAddrAdd()` configures the network IPv4 static parameters (IPv4 address, subnet mask and default gateway) required for the interface. More than one set of network parameters can be configured per interface. `NetIPv4_CfgAddrAdd()` can be repeated for as many network parameter sets as need configuring for an interface.

    IPv4 parameters can be added whenever as long as the interface was added (initialized) even if the interface is started or not.

    For Dynamic IPv4 configuration, μC/DHCPc is required

11. `NetIPv6_CfgAddrAdd()` configures the network IPv6 static parameters (IPv6 address and prefix length) required for the interface. More than one set of network parameters can be configured per interface. `NetIPv6_CfgAddrAdd()` can be repeated for as many network parameter sets as need configuring for an interface.

    IPv6 parameters can be added whenever as long as the interface is added (initialized) even if the interface is started or not.

    For the moment dynamic IPv6 is not yet supported either by IPv6 Autoconfig or DHCPv6c.

12. `NetIPv6_AddrAutoCfgHookSet()` is used to set the IPv6 Auto-Configuration hook function that will received the result of the Auto-Configuration process.

    a. The first argument is the Network interface number on which the Auto-Configuration will take place.

    b. The second argument is the pointer to the hook function the application needs to implement. Refer to section IPv6 Stateless Address Auto-Configuration Hook Function for an example.

13. `NetIPv6_AddrAutoCfgEn`() enables the interface to the IPv6 Address Auto-Configuration process. If the interface link state is already UP, the Auto-Configuration process will started immediately, else it will start after the interface link state becomes UP.

    a. The first argument is the Network interface number on which the Auto-Configuration will take place.

    b. The second argument enables or disables the Duplication Address Detection (DAD) during the Auto-Configuration process.

WiFi Sample Application

1. This example show how to initialize μC/TCP-IP:

    a. Initialize Stack tasks and objects

    b. Initialize a Wireless Interface

    c. Start that Wireless Interface

    d. Scan for Wireless network available

    e. Analyze scan results

    f. Join a Wireless network

    g. Configure IP addresses of that Interface

This example is based on template files so some modifications will be required, insert the appropriate project/board specific code to perform the stated actions. Note that the file `init_wifi.c`, located in the folder `$/Micrium/Software/uC-TCPIP/Examples/Init`, contains this sample application:

```
#include  <cpu_core.h>
#include  <lib_mem.h>
#include  <Source/net.h>
#include  <Source/net_ascii.h>
#include  <IF/net_if.h>
#include  <IF/net_if_wifi.h>
#ifdef NET_IPv4_MODULE_EN
#include  <IP/IPv4/net_ipv4.h>
#endif
#ifdef NET_IPv6_MODULE_EN
#include  <IP/IPv6/net_ipv6.h>
#endif
#include  <Cfg/Template/net_dev_cfg.h>                     /* See Note #1.           */
#include  <Dev/WiFi/Template/net_dev_wifi_template_spi.h>   /* See Note #2.           */
#include  <Dev/WiFi/Manager/Generic/net_wifi_mgr.h>
#include  <BSP/Template/net_bsp_wifi.h>                     /* See Note #3.           */

CPU_BOOLEAN  AppInit_TCPIP_WiFi (void)
{
#ifdef NET_IPv4_MODULE_EN
    NET_IPv4_ADDR       addr_ipv4;
    NET_IPv4_ADDR       msk_ipv4;
    NET_IPv4_ADDR       gateway_ipv4;
#endif
#ifdef NET_IPv6_MODULE_EN
    CPU_BOOLEAN         cfg_result;
#if (NET_IPv6_CFG_ADDR_AUTO_CFG_EN == DEF_DISABLED)
    NET_FLAGS           ipv6_flags;
    NET_IPv6_ADDR       addr_ipv6;
#endif
#endif
    NET_IF_NBR          if_nbr;
    NET_IF_WIFI_AP      ap[10];
    NET_IF_WIFI_SSID   *p_ssid;
    NET_IF_WIFI_SSID    ssid;
    NET_IF_WIFI_PSK     psk;
    CPU_INT16U          ctn;
    CPU_INT16U          i;
    CPU_INT16S          result;
    CPU_BOOLEAN         found;
    NET_ERR             err_net;

                                                    /* ------- PREREQUISITES MODULE INIT -------- */
    CPU_Init();                                     /* See Note #4.                               */
    Mem_Init();
                                                    /* ------ INIT NETWORK TASKS & OBJECTS ------ */
    err_net = Net_Init(&NetRxTaskCfg,               /* See Note #5.                               */
                       &NetTxDeallocTaskCfg,
                       &NetTmrTaskCfg);
    if (err_net != NET_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                    /* --------- ADD WIRELESS INTERFACE --------- */
                                                    /* See Note #6.                               */
    if_nbr = NetIF_Add(&NetIF_API_WiFi,             /* See Note #6b.                              */
                       &NetDev_API_TemplateWiFiSpi,     /* BSP API,              See Note #6d.        */
                       &NetDev_BSP_WiFi,            /* Device configuration, See Note #6e.        */
                       &NetDev_Cfg_WiFi_1,          /* PHY driver API,       See Note #6f.        */
                       &NetWiFiMgr_API_Generic,     /* PHY configuration,    See Note #6g.        */
                        DEF_NULL,
                       &err_net);
    if (err_net != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                    /* -------- START WIRELESS INTERFACE -------- */
                                                    /* See Note #7.                               */
    NetIF_Start(if_nbr, &err_net);                  /* Makes the IF ready to RX and TX.           */
    if (err_net != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                        /* ------- SCAN FOR WIRELESS NETWORKS ------- */
    ctn = NetIF_WiFi_Scan(if_nbr,                       /* See Note #11.                              */
                          ap,                           /* Access point table See Note #11a.          */
                          10,                           /* Access point table size.                   */
                          DEF_NULL,                     /* Hidden SSID See Note #11b.                 */
                          NET_IF_WIFI_CH_ALL,           /* Channel to scan See Note #11c.             */
                         &err_net);
    if (err_net != NET_IF_WIFI_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                        /* ------ ANALYSE WIFI NETWORKS FOUND ------- */
    found = DEF_NO;
    for (i = 0; i < ctn - 1; i++) {                     /* Browse table of access point found.        */
        p_ssid = &ap[i].SSID;
        result = Str_Cmp_N(p_ssid,                      /* Search for a specific WiFi Network SSID.   */
                          "Wifi_AP_SSID",               /* WiFi Network SSID.                         */
                           NET_IF_WIFI_STR_LEN_MAX_SSID);
        if (result == 0) {
            found = DEF_YES;
            break;
        }
    }
    if (found == DEF_NO) {
        return (DEF_FAIL);
    }
                                                        /* -------- JOIN A WIRELESS NETWORK --------- */
    Mem_Clr(&ssid, sizeof(ssid));
    Mem_Clr(&psk,  sizeof(psk));
    Str_Copy_N((CPU_CHAR *)&ssid,
                           "Wifi_AP_SSID",              /* WiFi Network SSID.                         */
                            12);                        /* SSID string length.                        */
    Str_Copy_N((CPU_CHAR *)&psk,
                           "Password",                  /* Pre shared Key (PSK), Wifi password.       */
                            8);                         /* PSK string length.                         */
    NetIF_WiFi_Join(if_nbr,                             /* See Note #12.                              */
                    ap[i].NetType,                      /* See Note #12a.                             */
                    NET_IF_WIFI_DATA_RATE_AUTO,         /* See Note #12b.                             */
                    ap[i].SecurityType,                 /* See Note #12c.                             */
                    NET_IF_WIFI_PWR_LEVEL_HI,           /* See Note #12d.                             */
                    ssid,                               /* See Note #12e.                             */
                    psk,                                /* See Note #12f.                             */
                   &err_net);
    if (err_net != NET_IF_WIFI_ERR_NONE) {
        return (DEF_FAIL);
    }

#ifdef NET_IPv4_MODULE_EN
                                                        /* ------ CONFIGURE IPV4 STATIC ADDR -------- */
                                                        /* For Dynamic IPv4 cfg, µC/DHCPc is required */
                                                        /* TODO Update IPv4 Addr following your ...   */
                                                        /* ... network requirements.                  */
                                                        /* See Note #10.                              */
    NetASCII_Str_to_IP("10.10.10.64",                   /* Convert Host IPv4 string to 32 bit addr.   */
                       &addr_ipv4,
                        NET_IPv4_ADDR_SIZE,
                       &err_net);
    NetASCII_Str_to_IP("255.255.255.0",                 /* Convert IPv4 mask string to 32 bit addr.   */
                       &msk_ipv4,
                        NET_IPv4_ADDR_SIZE,
                       &err_net);
    NetASCII_Str_to_IP("10.10.10.1",                    /* Convert Gateway string to 32 bit addr.     */
                       &gateway_ipv4,
                        NET_IPv4_ADDR_SIZE,
                       &err_net);
    NetIPv4_CfgAddrAdd(if_nbr,                          /* Add a statically-configured IPv4 host ...  */
                       addr_ipv4,                       /* ...  addr, subnet mask, & default     ...  */
                       msk_ipv4,                        /* ... gateway to the interface. See Note #11.*/
                       gateway_ipv4,
                      &err_net);
    if (err_net != NET_IPv4_ERR_NONE) {
        return (DEF_FAIL);
    }
#endif
#ifdef NET_IPv6_MODULE_EN
#if (NET_IPv6_CFG_ADDR_AUTO_CFG_EN == DEF_ENABLED)
                                                        /* ---- START IPV6 STATELESS AUTO-CONFIG ---- */
                                                        /* See Note #13.                              */
    NetIPv6_AddrAutoCfgHookSet(if_nbr,                  /* Set hook to received Auto-Cfg result.      */
                              &App_AutoCfgResult,       /* TODO update pointer to hook defined in App.*/
                              &err_net);

                                                        /* See Note #14.                              */
    cfg_result = NetIPv6_AddrAutoCfgEn(if_nbr,          /* Enable and Start Auto-Config process.      */
                                       DEF_YES,
                                      &err_net);
    if (cfg_result == DEF_FAIL) {
        return (DEF_FAIL);
    }

#else
                                                        /* ----- CFG IPV6 STATIC LINK LOCAL ADDR ---- */
                                                        /* DHCPv6c is not yet available.              */

    ipv6_flags = 0;
    DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_BLOCK_EN);    /* Set Address Configuration as blocking.     */
    DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_DAD_EN);      /* Enable DAD with Address Configuration.     */
                                                        /* TODO Update IPv6 Addr following your  ...  */
                                                        /* ... network requirements.                  */
                                                        /* See Note #10.                              */
    NetASCII_Str_to_IP("fe80::1111:1111",               /* Convert IPv6 string addr to 128 bit addr.  */
                       &addr_ipv6,
                        NET_IPv6_ADDR_SIZE,
                       &err_net);
    cfg_result = NetIPv6_CfgAddrAdd(if_nbr,             /* Add a statically-configured IPv6 host  ... */
                                   &addr_ipv6,          /* ... addr to the interface. See Note #12.   */
                                    64,
                                    ipv6_flags,
                                   &err_net);
    if (cfg_result == DEF_FAIL) {
        return (DEF_FAIL);
    }
#endif
#endif
    return (DEF_OK);
}
```

Listing - AppInit_TCPIP()

1. The device configuration template file should be copied to your application folder and modified to follow your requirements. Refer to the User's Manual for more information about how to configure your device.

   We recommend starting with a working configuration from an example project for your wireless module. Micriµm might have some projects available only for internal usage, so if no working project are found online, please ask at support@micrium.com for a configuration file example.

2. Most of the time Micriμm provides a Wireless device driver which can be found under the following folder:

   ```
   $/Micrium/Software/uC-TCPIP/Dev/WiFi/<module>/net_dev_<module>.h
   ```

   If Micrium does not support your network wireless device driver, you will have to write your own device driver starting from the Wireless Device driver template. Before starting to write your own driver, make sure that the driver is not already available.

3. The board support package (BSP) template file should be copied to your application folder and modified for your specific board. Refer to the User's Manual for more information about how to write a BSP.

   However we recommend, starting with a working configuration from an example project for your Wireless module. Micriμm might have some projects available only for internal usage, so if no working project are found online, please ask at support@micrium.com for a BSP file example specific for your MCU and your Wireless Module.

4. Some prerequisite module initializations are required. The following modules must be initialized prior to starting the Network Protocol stacks initialization:

   a. uC/CPU

   b. uC/LIB Memory module

5. Net_Init() is the Network Protocol stack's initialization function. It must only be called once and before any other Network functions.

   a. This function takes the three TCP-IP internal task configuration structures as arguments (such as priority, stack size, etc.). By default these configuration structures are defined in net_cfg.c :

```
NetRxTaskCfg             RX task configuration
NetTxDeallocTaskCfg      TX task configuration
NetTmrTaskCfg            Timer task configuration
```

b. We recommend you configure the Network Protocol Stack task priorities & Network application (such as a Web server) task priorities as follows:

```
NetTxDeallocTaskCfg                                 (highest priority)

Network applications (HTTPs, FTP, DNS, etc.)

NetTmrTaskCfg
NetRxTaskCfg                                        (lowest priority)
```

We recommend that the uC/TCP-IP Timer task and network interface Receive task be lower priority than almost all other application tasks; but we recommend that the network interface Transmit De-allocation task be higher priority than all application tasks that use uC/TCP-IP network services.

However, better performance can be observed when the Network applications are set with the lowest priority. Some experimentation could be required to identify the best task priority configuration.

6. `NetIF_Add()` is a network interface function responsible for initializing a Network Device driver.

a. `NetIF_Add()` returns the interface index number. The interface index number should start at '1', since the interface '0' is reserved for the loopback interface. The interface index number must be used when you want to access the interface using any Network interface API.

b. The first parameter is the address of the Network interface API. These API are provided by Micriµm and are defined in file `net_if_<type>.h`. It should be either:

```
NetIF_API_Ether Ethernet interface
```

`NetIF_API_WiFi Wireless interface`

c. The second parameter is the address of the device API function. The API should be defined in the Device driver header:

```
$/uC-TCPIP/Dev/<if_type>/<controller>/net_dev_<controller>.h
```

d. The third parameter is the address of the device BSP data structure. See Note #3 for more details.

e. The fourth parameter is the address of the device configuration data structure. See Note #1 for more details.

f. The fifth parameter is the address of the WiFi Manager API function. This API is provided by Micriµm and it's located in:

```
$/uC-TCPIP/Dev/WiFi/Manager/Generic/net_wifi_mgr.h
```

g. The sixth and last parameter is the address of the WiFi manager configuration data structure. Actually there are no configuration require with the generic WiFi manager. So this parameters can be kept as null.

7. `NetIF_Start()` makes the network interface scan, join or create adhoc wireless network. This function must interact with the wireless module thus some interrupt should be generated from the wireless's interrupt pin when calling this function.

8. `NetIF_WiFi_Scan()` is used to scan for available Wireless Network available in the range.

a. The second parameter is a table of access point that will be receive access points found in the range. Obviously, the maximum number of access point that the table can store must be past to the function.

b. It's possible to scan for a specific hidden network by passing a string that contains the SSID of the hidden network. If the scan request is for all access point, you only have to pass a null pointer.

c. The fourth parameter is the wireless channel to scan on, it can be:

```
NET_IF_WIFI_CH_ALL
NET_IF_WIFI_CH_1
NET_IF_WIFI_CH_2
NET_IF_WIFI_CH_3
NET_IF_WIFI_CH_4
NET_IF_WIFI_CH_5
NET_IF_WIFI_CH_6
NET_IF_WIFI_CH_7
NET_IF_WIFI_CH_8
NET_IF_WIFI_CH_9
NET_IF_WIFI_CH_10
NET_IF_WIFI_CH_11
NET_IF_WIFI_CH_12
NET_IF_WIFI_CH_13
NET_IF_WIFI_CH_14
```

9. `NetIF_WiFi_Join()` is used to join a wireless network. Note that the network must has been found during a scan previously. Once the wireless access point is join, it is possible to receive and transmit packet on the network.

a. The second parameter is the Network type it can be either:

```
NET_IF_WIFI_NET_TYPE_INFRASTRUCTURE
NET_IF_WIFI_NET_TYPE_ADHOC
```

The scan function should return the network type as well.

b. The third parameter is the wireless date rate to configure:

```
NET_IF_WIFI_DATA_RATE_AUTO
NET_IF_WIFI_DATA_RATE_1_MBPS
NET_IF_WIFI_DATA_RATE_2_MBPS
```

```
NET_IF_WIFI_DATA_RATE_5_5_MBPS
NET_IF_WIFI_DATA_RATE_6_MBPS
NET_IF_WIFI_DATA_RATE_9_MBPS
NET_IF_WIFI_DATA_RATE_11_MBPS
NET_IF_WIFI_DATA_RATE_12_MBPS
NET_IF_WIFI_DATA_RATE_18_MBPS
NET_IF_WIFI_DATA_RATE_24_MBPS
NET_IF_WIFI_DATA_RATE_36_MBPS
NET_IF_WIFI_DATA_RATE_48_MBPS
NET_IF_WIFI_DATA_RATE_54_MBPS
NET_IF_WIFI_DATA_RATE_MCS0
NET_IF_WIFI_DATA_RATE_MCS1
NET_IF_WIFI_DATA_RATE_MCS2
NET_IF_WIFI_DATA_RATE_MCS3
NET_IF_WIFI_DATA_RATE_MCS4
NET_IF_WIFI_DATA_RATE_MCS5
NET_IF_WIFI_DATA_RATE_MCS6
NET_IF_WIFI_DATA_RATE_MCS7
NET_IF_WIFI_DATA_RATE_MCS8
NET_IF_WIFI_DATA_RATE_MCS9
NET_IF_WIFI_DATA_RATE_MCS10
NET_IF_WIFI_DATA_RATE_MCS11
NET_IF_WIFI_DATA_RATE_MCS12
NET_IF_WIFI_DATA_RATE_MCS13
NET_IF_WIFI_DATA_RATE_MCS14
NET_IF_WIFI_DATA_RATE_MCS15
```

c. The fourth parameter is the wireless network's security type. It can be:

```
  NET_IF_WIFI_SECURITY_OPEN
 NET_IF_WIFI_SECURITY_WEP
 NET_IF_WIFI_SECURITY_WPA
 NET_IF_WIFI_SECURITY_WPA2
```

The scan function should return the network security type as well.

d. The fifth parameter is the wireless radio's power level. It can be:

```
 NET_IF_WIFI_PWR_LEVEL_LO
 NET_IF_WIFI_PWR_LEVEL_MED
 NET_IF_WIFI_PWR_LEVEL_HI
```

e. The sixth parameter is the access point's SSID to join.

f. The seventh parameter is the Pre shared key (PSK) of the access point's. If the security type of the access point is open, the PSK can set to null.

10. `NetASCII_Str_to_IP()` converts the human readable address into a format required by the protocol stack.

In this example the IP address used, 10.10.10.64, addresses out of the 10.10.10.1 network with a subnet mask of 255.255.255.0. To match different networks, the IP address, the subnet mask and the default gateway's IP address have to be customized.

11. `NetIPv4_CfgAddrAdd()` configures the network IPv4 static parameters (IPv4 address, subnet mask and default gateway) required for the interface. More than one set of network parameters can be configured per interface. `NetIPv4_CfgAddrAdd()` can be repeated for as many network parameter sets as need configuring for an interface.

    IPv4 parameters can be added whenever as long as the interface was added (initialized) even if the interface is started or not.

    For Dynamic IPv4 configuration, μC/DHCPc is required

12. `NetIPv6_CfgAddrAdd()` configures the network IPv6 static parameters (IPv6 address and prefix length) required for the interface. More than one set of network parameters can be configured per interface. `NetIPv6_CfgAddrAdd()` can be repeated for as many network parameter sets as need configuring for an interface.

    IPv6 parameters can be added whenever as long as the interface is added (initialized) even if the interface is started or not.

    For the moment dynamic IPv6 is not yet supported either by IPv6 Autoconfig or DHCPv6c.

13. `NetIPv6_AddrAutoCfgHookSet()` is used to set the IPv6 Auto-Configuration hook function that will received the result of the Auto-Configuration process.

    a. The first argument is the Network interface number on which the Auto-Configuration will take place.

    b. The second argument is the pointer to the hook function the application needs to implement. Refer to section IPv6 Stateless Address Auto-Configuration Hook Function for an example.

14. `NetIPv6_AddrAutoCfgEn`() enables the interface to the IPv6 Address Auto-Configuration process. If the interface link state is already UP, the Auto-Configuration process will started immediately, else it will start after the interface link state becomes UP.

    a. The first argument is the Network interface number on which the Auto-Configuration will take place.

    b. The second argument enables or disables the Duplication Address Detection (DAD) during the Auto-Configuration process.

Multiple Interfaces Sample Application

1. This example show how to initialize μC/TCP-IP with multiple interface:

    a. Initialize Stack tasks and objects

    b. Initialize an Ethernet Interface

    c. Start that Ethernet Interface

    d. Configure IP addresses of the Ethernet Interface

    e. Initialize an Wireless Interface

    f. Start that Wireless Interface

    g. Scan for Wireless networks available

    h. Analyze scan result

    i. Join a Wireless network

    j. Configure IP addresses of that Wireless Interface

This example is based on template files so some modifications will be required, insert the appropriate project/board specific code to perform the stated actions. Note that the file `init_multiple_if.c`, located in the folder `$/Micrium/Software/uC-TCPIP/Examples/Init`, contains this sample application:

```
#include  <cpu_core.h>
#include  <lib_mem.h>
#include  <Source/net.h>
#include  <Source/net_ascii.h>
#include  <IF/net_if.h>
#include  <IF/net_if_wifi.h>
#ifdef NET_IPv4_MODULE_EN
#include  <IP/IPv4/net_ipv4.h>
#endif
#ifdef NET_IPv6_MODULE_EN
#include  <IP/IPv6/net_ipv6.h>
#endif
#include  <Cfg/Template/net_dev_cfg.h>                      /* Device configuration header.           */
#include  <Dev/Ether/Template/net_dev_ether_template_dma.h> /* Device driver header.                  */
#include  <Dev/WiFi/Template/net_dev_wifi_template_spi.h>   /* Device driver header.                  */
#include  <Dev/Ether/PHY/Generic/net_phy.h>                 /* PHY driver header.                     */
```

```
#include  <Dev/WiFi/Manager/Generic/net_wifi_mgr.h>
#include  <BSP/Template/net_bsp_ether.h>                    /* BSP header.                            */
#include  <BSP/Template/net_bsp_wifi.h>                     /* BSP header.                            */


CPU_BOOLEAN  AppInit_TCPIP_MultipleIF (void)
{
#ifdef NET_IPv4_MODULE_EN
    NET_IPv4_ADDR       addr_ipv4;
    NET_IPv4_ADDR       msk_ipv4;
    NET_IPv4_ADDR       gateway_ipv4;
#endif
#ifdef NET_IPv6_MODULE_EN
    CPU_BOOLEAN         cfg_result;
#if (NET_IPv6_CFG_ADDR_AUTO_CFG_EN == DEF_DISABLED)
    NET_FLAGS           ipv6_flags;
    NET_IPv6_ADDR       addr_ipv6;
#endif
#endif
    NET_IF_NBR          if_nbr_ether;
    NET_IF_NBR          if_nbr_wifi;
    NET_IF_WIFI_AP      ap[10];
    NET_IF_WIFI_SSID   *p_ssid;
    NET_IF_WIFI_SSID    ssid;
    NET_IF_WIFI_PSK     psk;
    CPU_INT16U          ctn;
    CPU_INT16U          i;
    CPU_INT16S          result;
    CPU_BOOLEAN         found;
    NET_ERR             err_net;
                                                            /* ---- INIT NETWORK TASKS & OBJECTS ---- */
    err_net = Net_Init(&NetRxTaskCfg,
                       &NetTxDeallocTaskCfg,
                       &NetTmrTaskCfg);
    if (err_net != NET_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                            /* ------- ADD ETHERNET INTERFACE ------- */
    if_nbr_ether = NetIF_Add(&NetIF_API_Ether,
                             &NetDev_API_TemplateEtherDMA,  /* Device driver API                     */
                             &NetDev_BSP_BoardDev_Nbr,      /* BSP API                               */
                             &NetDev_Cfg_Ether_1,           /* Device configuration                  */
                             &NetPhy_API_Generic,           /* PHY driver API                        */
                             &NetPhy_Cfg_Ether_1,           /* PHY configuration                     */
                             &err_net);
    if (err_net != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                            /* ------ START ETHERNET INTERFACE ------ */
    NetIF_Start(if_nbr_ether, &err_net);                    /* Makes the IF ready to RX and TX.       */
    if (err_net != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }
#ifdef NET_IPv4_MODULE_EN
                                                            /* ------- CFG IPV4 STATIC ADDR --------- */
                                                            /* For Dynamic IPv4 cfg, DHCPc is required*/
                                                            /* Update IPv4 Addr following your  ...   */
                                                            /* ... network requirements.              */
    NetASCII_Str_to_IP("10.10.10.64",                       /* Convert IPv4 string addr to 32 bit addr*/
                       &addr_ipv4,
                        NET_IPv4_ADDR_SIZE,
                       &err_net);
    NetASCII_Str_to_IP("255.255.255.0",                     /* Convert IPv4 mask to 32 bit addr.      */
                       &msk_ipv4,
                        NET_IPv4_ADDR_SIZE,
                       &err_net);
    NetASCII_Str_to_IP("10.10.10.1",                        /* Convert Gateway string to 32 bit addr. */
                       &gateway_ipv4,
                         NET_IPv4_ADDR_SIZE,
                        &err_net);
    NetIPv4_CfgAddrAdd(if_nbr_ether,                        /* Add a statically-configured IPv4  ... */
                        addr_ipv4,                          /* ... host addr, subnet mask, &     ... */
                        msk_ipv4,                           /* ... default gateway to the IF.        */
                        gateway_ipv4,
                       &err_net);
    if (err_net != NET_IPv4_ERR_NONE) {
        return (DEF_FAIL);
    }
#endif
#ifdef NET_IPv6_MODULE_EN
#if (NET_IPv6_CFG_ADDR_AUTO_CFG_EN == DEF_ENABLED)
                                                            /* --- START IPV6 STATELESS AUTO-CFG ---- */
    NetIPv6_AddrAutoCfgHookSet(if_nbr_ether,                /* Set Hook to received Auto-Cfg result.  */
                               &App_AutoCfgResult,
                               &err_net);

    cfg_result = NetIPv6_AddrAutoCfgEn(if_nbr_ether,        /* Enable and Start Auto-Cfg process.     */
                                        DEF_YES,
                                       &err_net);
    if (cfg_result == DEF_FAIL) {
        return (DEF_FAIL);
    }

#else
                                                            /* --- CFG IPV6 STATIC LINK LOCAL ADDR -- */
                                                            /* DHCPv6c is not yet available.          */
                                                            /* TODO Update IPv6 Addr following your...*/
                                                            /* ... network requirements.              */
    NetASCII_Str_to_IP("fe80::1111:1111",                   /* Convert IPv6 string to 128 bit addr.   */
                        &addr_ipv6,
                         NET_IPv6_ADDR_SIZE,
                        &err_net);

    ipv6_flags = 0;
    DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_BLOCK_EN);        /* Set Addr Cfg as blocking.              */
    DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_DAD_EN);          /* Enable DAD with Addr Configuration.    */

    cfg_result = NetIPv6_CfgAddrAdd(if_nbr_ether,           /* Add a statically-configured IPv6 ...   */
                                    &addr_ipv6,             /* ...  host address to the interface.    */
                                     64,
                                     ipv6_flags,
                                    &err_net);
    if (cfg_result == DEF_FAIL) {
        return (DEF_FAIL);
    }
#endif
#endif
                                                            /* ------- ADD WIRELESS INTERFACE ------- */
    if_nbr_wifi = NetIF_Add(&NetIF_API_WiFi,
                            &NetDev_API_TemplateWiFiSpi,    /* Change following your Device driver API*/
                            &NetDev_BSP_WiFi,               /* Change for your BSP API.               */
                            &NetDev_Cfg_WiFi_1,             /* Change for Device configuration.       */
                            &NetWiFiMgr_API_Generic,
                             DEF_NULL,
                            &err_net);
    if (err_net != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                            /* ------ START WIRELESS INTERFACE ------ */
    NetIF_Start(if_nbr_wifi, &err_net);                     /* Makes the IF ready to RX and TX.       */
    if (err_net != NET_IF_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                            /* ----- SCAN FOR WIRELESS NETWORKS ----- */
    ctn = NetIF_WiFi_Scan(if_nbr_wifi,
                          ap,                               /* Access point table.                    */
                        10,                             /* Access point table size.              */
                        DEF_NULL,                       /* Hidden SSID.                          */
                        NET_IF_WIFI_CH_ALL,             /* Channel to scan.                      */
                       &err_net);
    if (err_net != NET_IF_WIFI_ERR_NONE) {
        return (DEF_FAIL);
    }
                                                        /* --- ANALYSE WIRELESS NETWORKS FOUND -- */
    found = DEF_NO;
    for (i = 0; i < ctn - 1; i++) {                     /* Browse table of access point found.   */
        p_ssid = &ap[i].SSID;
        result = Str_Cmp_N((CPU_CHAR *)p_ssid,              /* Search for a specific WiFi Network SSID*/
                              "Wifi_AP_SSID",           /* Change for your WiFi Network SSID.    */
                               NET_IF_WIFI_STR_LEN_MAX_SSID);
        if (result == 0) {
            found = DEF_YES;
            break;
        }
    }
    if (found == DEF_NO) {
        return (DEF_FAIL);
    }
                                                        /* ------ JOIN A WIRELESS NETWORK ------- */
    Mem_Clr(&ssid, sizeof(ssid));
    Mem_Clr(&psk,  sizeof(psk));
    Str_Copy_N((CPU_CHAR *)&ssid,
                           "Wifi_AP_SSID",              /* Change for your WiFi Network SSID.    */
                            12);                        /* SSID string length.                   */
    Str_Copy_N((CPU_CHAR *)&psk,
                           "Password",                  /* Change for your WiFi Network Password. */
                            8);                         /* PSK string length.                    */
    NetIF_WiFi_Join(if_nbr_wifi,
                    ap[i].NetType,                      /* WiFi Network type.                    */
                    NET_IF_WIFI_DATA_RATE_AUTO,         /* Data rate.                            */
                    ap[i].SecurityType,                 /* WiFi Network security type.           */
                    NET_IF_WIFI_PWR_LEVEL_HI,           /* Power level.                          */
                    ssid,                               /* WiFi Network SSID.                    */
                    psk,                                /* WiFi Network PSK.                     */
                   &err_net);
    if (err_net != NET_IF_WIFI_ERR_NONE) {
        return (DEF_FAIL);
    }
#ifdef NET_IPv4_MODULE_EN
                                                        /* -------- CFG IPV4 STATIC ADDR -------- */
                                                        /* For Dynamic IPv4 cfg, DHCPc is required*/
                                                        /* Update IPv4 Addr following your  ...  */
                                                        /* ... network requirements.             */
    NetASCII_Str_to_IP("192.168.1.10",                  /* Convert IPv4 string addr to 32 bit addr*/
                       &addr_ipv4,
                        NET_IPv4_ADDR_SIZE,
                       &err_net);
    NetASCII_Str_to_IP("255.255.255.0",                 /* Convert Mask string to 32 bit addr.   */
                       &msk_ipv4,
                        NET_IPv4_ADDR_SIZE,
                       &err_net);
    NetASCII_Str_to_IP("192.168.1.1",                   /* Convert Gateway string to 32 bit addr. */
                       &gateway_ipv4,
                        NET_IPv4_ADDR_SIZE,
                       &err_net);
    NetIPv4_CfgAddrAdd(if_nbr_wifi,                     /* Add a statically-configured IPv4 ...  */
                       addr_ipv4,                       /* ... host addr, subnet mask, &    ...  */
                       msk_ipv4,                        /* ... default gateway to the IF.        */
                       gateway_ipv4,
                      &err_net);
    if (err_net != NET_IPv4_ERR_NONE) {
        return (DEF_FAIL);
    }
#endif
```

```
#ifdef NET_IPv6_MODULE_EN
#if (NET_IPv6_CFG_ADDR_AUTO_CFG_EN == DEF_ENABLED)
                                                            /* ---- START IPV6 STATELESS AUTO-CFG --- */
    NetIPv6_AddrAutoCfgHookSet(if_nbr_wifi,                 /* Set hook to received Auto-Cfg result.  */
                               &App_AutoCfgResult,           /* TODO update ptr to hook defined in
App.*/
                               &err_net);

    cfg_result = NetIPv6_AddrAutoCfgEn(if_nbr_wifi,         /* Enable and Start Auto-Cfg process.     */
                                        DEF_YES,
                                       &err_net);
    if (cfg_result == DEF_FAIL) {
        return (DEF_FAIL);
    }

#else
                                                            /* --- CFG IPV6 STATIC LINK LOCAL ADDR -- */
                                                            /* DHCPv6c is not yet available.          */
                                                            /* Update IPv6 Addr following your ...    */
                                                            /* ... network requirements.              */

    NetASCII_Str_to_IP("fe80::4444:1111",                   /* Convert IPv6 string to 128 bit addr.   */
                       &addr_ipv6,
                        NET_IPv6_ADDR_SIZE,
                       &err_net);

    ipv6_flags = 0;
    DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_BLOCK_EN);        /* Set Address Configuration as blocking. */
    DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_DAD_EN);          /* Enable DAD with Address Configuration. */

    cfg_result = NetIPv6_CfgAddrAdd(if_nbr_wifi,            /* Add a statically-configured IPv6 ...   */
                                   &addr_ipv6,              /* ... host address to the interface.     */
                                    64,
                                    ipv6_flags,
                                   &err_net);
    if (cfg_result == DEF_FAIL) {
        return (DEF_FAIL);
    }
#endif
    return (DEF_OK);
}
```

Refer to the sample codes in section Ethernet Sample Application and WiFi Sample Application for more details on the diverse function calls.

Network Board Support Package

This chapter describes all board-specific functions that you may need to implement.

In order for a device driver to be platform independent, it is necessary to provide a layer of code that abstracts details such as configuring clocks, interrupt controllers, general-purpose input/ouput (GPIO) pins, direct-memory access (DMA) modules, and other such hardware modules. The board support package (BSP) code layer enables you to implement certain high-level functionality in μC/TCP-IP that is independent of any specific hardware. It also allows you to reuse device drivers from various architectures and bus configurations without having to customize μC/TCP-IP or the device driver source code for each architecture or hardware platform.

To understand the concepts discussed in this guide, you should be familiar with networking principles, the TCP/IP stack, real-time operating systems, microcontrollers and processors.

Micrium provides template BSP files (net_bsp_ether.c/h and net_bsp_wifi.c/h), which should be copied to your project and modified depending on the combination of compiler, processor, board and device hardware used. However, Micrium might have BSP available for some specific Evaluation board which are not delivered. So before starting to write your own BSP, you can ask Micrium for a working sample code for a MCU, if a sample code is available you could just apply minor modification to be compatible with your compiler and board.

- Ethernet BSP Layer
- Wireless BSP Layer
- Specifying the Interface Number of the Device ISR

Ethernet BSP Layer

Description of the Ethernet BSP API

This section describes the BSP API functions that you should implement during the integration of an Ethernet interface for µC/TCP-IP.

For each Ethernet interface/device, an application must implement in `net_bsp_ether.c`, a unique device-specific implementation of each of the following BSP functions:

```
void NetDev_CfgClk          (NET_IF  *p_if,
                             NET_ERR *p_err);
void NetDev_CfgIntCtrl      (NET_IF  *p_if,
                             NET_ERR *p_err);
void NetDev_CfgGPIO         (NET_IF  *p_if,
                             NET_ERR *p_err);
CPU_INT32U NetDev_ClkFreqGet(NET_IF  *p_if,
                             NET_ERR *p_err);
```

Since each of these functions is called from a unique instantiation of its corresponding device driver, a pointer to the corresponding network interface (`p_if`) is passed in order to access the specific interface's device configuration or data.

Network device driver BSP functions may be arbitrarily named but since development boards with multiple devices require unique BSP functions for each device, it is recommended that each device's BSP functions be named using the following convention:

`NetDev_[Device]<Function>[Number]()`

`[Device]`

Network device name or type. For example, MACB. (Optional if the development board does not support multiple devices.)

`<Function>`

Network device BSP function. For example, `CfgClk`.

`[Number]`

Network device number for each specific instance of device (optional if the development board does not support multiple instances of the specific device)

For example, the `NetDev_CfgClk()` function for the #2 MACB Ethernet controller on an Atmel AT91SAM9263-EK should be named `NetDev_MACB_CfgClk2()`, or `NetDev_MACB_CfgClk_2()` with additional underscore optional.

Similarly, network devices’ BSP-level interrupt service routine (ISR) handlers should be named using the following convention:

```
NetDev_[Device]ISR_Handler[Type][Number]()
```

`[Device]`

Network device name or type. For example, MACB. (Optional if the development board does not support multiple devices.)

`[Type]`

Network device interrupt type. For example, receive interrupt. (Optional if interrupt type is generic or unknown.)

`[Number]`

Network device number for each specific instance of device (optional if the development board does not support multiple instances of a specific device).

For example, the receive ISR handler for the #2 MACB Ethernet controller on an Atmel AT91SAM9263-EK should be named `NetDev_MACB_ISR_HandlerRx2()`, or `NetDev_MACB_ISR_HandlerRx_2()`, with additional underscore optional.

Next, the BSP functions for each device/interface must be organized into an interface structure. This structure is used by the device driver to call specific devices’ BSP functions via function pointer instead of by name. It allows applications to add, initialize, and configure any number of instances of various devices and drivers by creating similar but unique BSP functions and interface structures for each network device/interface. (See section Interface Programming for details on how applications add interfaces to µC/TCP-IP.)

The BSP for each device or interface must be declared in the BSP source file (`net_bsp.c`) for each application or development board. The BSP must also be externally declared in the network BSP header file (`net_bsp.h`) with exactly the same name and type as declared in `net_bsp.c`. These BSP interface structures and their corresponding functions must have unique names, and should clearly identify the development board, device name, function name, and possibly the specific device number (assuming the development board supports multiple instances of any given device). BSP interface structures may be given arbitrary names, but it is recommended that they be named using the following convention:

`NetDev_BSP_<Board><Device>[Number]{}`

`<Board>`

Development board name. For example, Atmel AT91SAM9263-EK).

`<Device>`

Network device name or type. For example, MACB.

`[Number]`

Network device number for each specific instance of the device (optional if the development board does not support multiple instances of the device).

For example, a BSP interface structure for the #2 MACB Ethernet controller on an Atmel AT91SAM9263-EK board should be named `NetDev_BSP_AT91SAM9263-EK_MACB_2{}` and declared in the AT91SAM9263-EK board's `net_bsp.c`:

```
/* AT91SAM9263-EK MACB #2's BSP fnct ptrs : */
const NET_DEV_BSP_ETHER NetDev_BSP_AT91SAM9263-EK_MACB_2 = {
                                                              NetDev_MACB_CfgClk_2,            /* Cfg
MACB #2's clk(s)      */
                                                              NetDev_MACB_CfgIntCtrl_2,        /* Cfg
MACB #2's int ctrl(s) */
                                                              NetDev_MACB_CfgGPIO_2,           /* Cfg
MACB #2's GPIO        */
                                                              NetDev_MACB_ClkFreqGet_2         /* Get
MACB #2's clk freq    */
                                                            };
```

In order for the application to configure an interface with this BSP interface structure, the structure must also be externally declared in the AT91SAM9263-EK board's `net_bsp.h`:

```
extern const NET_DEV_BSP_ETHER NetDev_BSP_AT91SAM9263-EK_MACB_2;
```

Lastly, the AT91SAM9263-EK board’s MACB #2 BSP functions must also be declared in `net_bsp.c`:

```
static void NetDev_MACB_CfgClk_2         (NET_IF  *p_if,
                                          NET_ERR *p_err);
static void NetDev_MACB_CfgIntCtrl_2     (NET_IF  *p_if,
                                          NET_ERR *p_err);
static void NetDev_MACB_CfgGPIO_2        (NET_IF  *p_if,
                                          NET_ERR *p_err);
static CPU_INT32U NetDev_MACB_ClkFreqGet_2(NET_IF  *p_if,
                                          NET_ERR *p_err);
```

Note that since all network device BSP functions are accessed only by function pointer via their corresponding BSP interface structure, they don’t need to be globally available and should therefore be declared as `static` .

Also note that although certain device drivers may not need to implement or call all of the above network device BSP functions, we recommend that each device’s BSP interface structure define all device BSP functions, and not assign any of its function pointers to `NULL`. Instead, for any device’s unused BSP functions, create empty functions that return `NET_DEV_ERR_NONE`. This way, if the device driver is ever modified to start using a previously unused BSP function, there will at least be an empty function for the BSP function pointer to execute.

Details for these functions may be found in their respective sections in Ethernet Device BSP Functions and templates for network device BSP functions and BSP interface structures are available in the `\Micrium\Software\uC-TCPIP\BSP\Template\` directories.

Configuring Clocks for an Ethernet Device

`NetDev_CfgClk()` sets a specific network device’s clocks to a specific interface.

Each network device’s `NetDev_CfgClk()` should configure and enable all required clocks for the specified network device. For example, on some devices it may be necessary to enable clock gating for an embedded Ethernet MAC, as well as various GPIO modules in order to configure Ethernet PHY pins for (R)MII mode and interrupts. See function NetDev_CfgClk for more information.

Configuring General I/O for an Ethernet Device

`NetDev_CfgGPIO()` configures a specific network device's general-purpose input/output (GPIO) on a specific interface. This function is called by a device driver's `NetDev_Init()`.

Each network device's `NetDev_CfgGPIO()` should configure all required GPIO pins for the network device. For Ethernet devices, this function is necessary to configure the (R)MII bus pins, depending on whether the user has configured an Ethernet interface to operate in the RMII or MII mode, and optionally the Ethernet PHY interrupt pin.

See function NetDev_CfgGPIO for more information.

Configuring the Interrupt Controller for an Ethernet Device

`NetDev_CfgIntCtrl()` is called by a device driver's `NetDev_Init()` to configure a specific network device's interrupts and/or interrupt controller on a specific interface.

Each network device's `NetDev_CfgIntCtrl()` function must configure and enable all required interrupt sources for the network device. This means it must configure the interrupt vector address of each corresponding network device BSP interrupt service routine (ISR) handler and enable its corresponding interrupt source.

For `NetDev_CfgIntCtrl()`, the following actions should be performed:

1. Configure/store each device's network interface number to be available for all necessary `NetDev_ISR_Handler()` functions (see section Specifying the Interface Number of the Device ISR for more information). Even though devices are added dynamically, the device's interface number must be saved in order for each device's ISR handlers to call `NetIF_ISR_Handler()` with the device's network interface number.

2. Configure each of the device's interrupts on an interrupt controller (either an external or CPU-integrated interrupt controller). However, vectored interrupt controllers may not require higher-level interrupt controller sources to be explicitly configured and enabled. In this case, you may need to configure the system's interrupt vector table with the name of the ISR handler functions declared in `net_bsp.c`.

`NetDev_CfgIntCtrl()` should enable only each devices' interrupt sources, but not the local device-level interrupts themselves, which are enabled by the device driver only after the device has been fully configured and started.

See function NetDev_CfgIntCtrl for more information.

Getting a Device Clock Frequency

`NetDev_ClkFreqGet()` returns a specific network device's clock frequency for a specific interface. This function is called by a device driver's `NetDev_Init()`.

Each network device's `NetDev_ClkFreqGet()` should return the device's clock frequency (in Hz). For Ethernet devices, this is the clock frequency of the device's (R)MII bus. The device driver's `NetDev_Init()` uses the returned clock frequency to configure an appropriate bus divider to ensure that the (R)MII bus logic operates within an allowable range. In general, the device driver should not configure the divider such that the (R)MII bus operates faster than 2.5MHz.

See function NetDev_ClkGetFreq for more information.

Wireless BSP Layer

Description of the Wireless BSP API

This section describes the BSP API functions that you should implement during the integration of a wireless interface for µC/TCP-IP.

For each wireless interface/device, an application must implement (in `net_bsp_wifi.c`) a unique device-specific implementation of each of the following BSP functions:

```
void NetDev_WiFi_Start        (NET_IF                          *p_if,
                               NET_ERR                         *p_err);

void NetDev_WiFi_Stop         (NET_IF                          *p_if,
                               NET_ERR                         *p_err);

void NetDev_WiFi_CfgGPIO      (NET_IF                          *p_if,
                               NET_ERR                         *p_err);

void NetDev_WiFi_CfgIntCtrl   (NET_IF                          *p_if,
                               NET_ERR                         *p_err);

void NetDev_WiFi_IntCtrl      (NET_IF                          *p_if,
                               CPU_BOOLEAN                      en,
                               NET_ERR                         *p_err);

void NetDev_WiFi_SPI_Init     (NET_IF                          *p_if,
                               NET_ERR                         *p_err);

void NetDev_WiFi_SPI_Lock     (NET_IF                          *p_if,
                               NET_ERR                         *p_err);

void NetDev_WiFi_SPI_Unlock   (NET_IF                          *p_if);

void NetDev_WiFi_SPI_WrRd     (NET_IF                          *p_if,
                               CPU_INT08U                      *p_buf_wr,
                               CPU_INT08U                      *p_buf_rd,
                               CPU_INT16U                       len,
                               NET_ERR                         *p_err);

void NetDev_WiFi_SPI_ChipSelEn (NET_IF                         *p_if,
                               NET_ERR                         *p_err);

void NetDev_WiFi_SPI_ChipSelDis(NET_IF                         *p_if);

void NetDev_WiFi_SPI_Cfg      (NET_IF                          *p_if,
                               NET_DEV_CFG_SPI_CLK_FREQ         freq,
                               NET_DEV_CFG_SPI_CLK_POL          pol,
                               NET_DEV_CFG_SPI_CLK_PHASE        phase,
                               NET_DEV_CFG_SPI_XFER_UNIT_LEN    xfer_unit_len,
                               NET_DEV_CFG_SPI_XFER_SHIFT_DIR   xfer_shift_dir,
                               NET_ERR                         *p_err);
```

Since each of these functions is called from a unique instantiation of its corresponding device driver, a pointer to the corresponding network interface (`p_if`) is passed in order to access the specific interface's device configuration or data.

Network device driver BSP functions may be arbitrarily named but since development boards with multiple devices require unique BSP functions for each device, it is recommended that each device's BSP functions be named using the following convention:

`NetDev_[Device]<Function>[Number]()`

`[Device]`

Network device name or type. For example, MACB (optional if the development board does not support multiple devices).

`<Function>`

Network device BSP function. For example, `CfgClk`

`[Number]`

Network device number for each specific instance of device (optional if the development board does not support multiple instances of a specific device)

For example, the `NetDev_CfgGPIO()` function for the #2 RS9110-N-21 wireless module on an Atmel AT91SAM9263-EK should be named `NetDev_RS9110N21_CfgGPIO2()`, or `NetDev_RS9110N21_CfgGPIO_2()` with additional underscore optional.

Similarly, network devices' BSP-level interrupt service routine (ISR) handlers should be named using the following convention:

`NetDev_[Device]ISR_Handler[Type][Number]()`

`[Device]`

Network device name or type. For example, MACB. (Optional if the development board does not support multiple devices.)

`[Type]`

Network device interrupt type. For example, receive interrupt. (Optional if interrupt type is generic or unknown.)

`[Number]`

Network device number for each specific instance of device (optional if the development board does not support multiple instances of a specific device).

For example, the receive ISR handler for the #2 RS9110-N-21 wireless module on an Atmel AT91SAM9263-EK should be named `NetDev_RS9110N21_ISR_HandlerRx2()`, or `NetDev_RS9110N21_ISR_HandlerRx_2()` with additional underscore optional.

Next, each device's/interface's BSP functions must be organized into an interface structure used by the device driver to call specific devices' BSP functions via function pointer instead of by name. This allows applications to add, initialize, and configure any number of instances of various devices and drivers by creating similar but unique BSP functions and interface structures for each network device/interface. (See section Interface Programming for details on how applications add interfaces to µC/TCP-IP.)

Each device's/interface's BSP interface structure must be declared in the application's/ development board's network BSP source file, `net_bsp.c`, as well as externally declared in the network BSP header file, `net_bsp.h`, with the exact same name and type as declared in `net_bsp.c`. These BSP interface structures and their corresponding functions must be uniquely named and should clearly identify the development board, device name, function name, and possibly the specific device number (assuming the development board supports multiple instances of any given device). BSP interface structures may be arbitrarily named but it is recommended that they be named using the following convention:

```
NetDev_BSP_<Board><Device>[Number]{}
```

`<Board>`

Development board name. For example, Atmel AT91SAM9263-EK.

`<Device>`

Network device name (or type). For example, RS9110-N-21.

`[Number]`

Network device number for each specific instance of the device (optional if the development board does not support multiple instances of the device).

For example, a BSP interface structure for the #2 RS9110-N21 wireless module on an Atmel AT91SAM9263-EK board should be named `NetDev_BSP_AT91SAM9263-EK_RS9110N21_2{}` and declared in the AT91SAM9263-EK board's `net_bsp.c`:

```
/* AT91SAM9263-EK RS9110-N21 #2's BSP fnct ptrs : */
const NET_DEV_BSP_WIFI_SPI NetDev_BSP_AT91SAM9263-EK_RS9110N21_2 = {
                                                                      NetDev_RS9110N21_Start_2
                                                                      NetDev_RS9110N21_Stop_2,
                                                                      NetDev_RS9110N21_CfgGPIO_2,
                                                                      NetDev_RS9110N21_CfgExtIntCtrl_2
                                                                      NetDev_RS9110N21_ExtIntCtrl_2,
                                                                      NetDev_RS9110N21_SPI_Cfg_2,
                                                                      NetDev_RS9110N21_SPI_Lock_2,
                                                                      NetDev_RS9110N21_SPI_Unlock_2,
                                                                      NetDev_RS9110N21_SPI_WrRd_2,
                                                                      NetDev_RS9110N21_SPI_ChipSelEn_2,
                                                                      NetDev_RS9110N21_SPI_ChipSelDis_2,
                                                                      NetDev_RS9110N21_SetCfg_2
                                                                    };
```

And in order for the application to configure an interface with this BSP interface structure, the structure must be externally declared in the AT91SAM9263-EK board's `net_bsp.h` :

```
extern const NET_DEV_BSP_WIFI_SPI NetDev_BSP_AT91SAM9263-EK_RS9110N21_2;
```

Lastly, the board's RS9110-N-21 #2 BSP functions must also be declared in `net_bsp.c`:

```
static void NetDev_RS9110N21_Start_2        (NET_IF                          *p_if,
                                             NET_ERR                         *p_err);

static void NetDev_RS9110N21_Stop_2         (NET_IF                          *p_if,
                                             NET_ERR                         *p_err);

static void NetDev_RS9110N21_CfgGPIO_2      (NET_IF                          *p_if,
                                             NET_ERR                         *p_err);

static void NetDev_RS9110N21_CfgIntCtrl_2   (NET_IF                          *p_if,
                                             NET_ERR                         *p_err);

static void NetDev_RS9110N21_IntCtrl_2      (NET_IF                          *p_if,
                                             CPU_BOOLEAN                      en,
                                             NET_ERR                         *p_err);

static void NetDev_RS9110N21_SPI_Init_2     (NET_IF                          *p_if,
                                             NET_ERR                         *p_err);

static void NetDev_RS9110N21_SPI_Lock_2     (NET_IF                          *p_if,
                                             NET_ERR                         *p_err);

static void NetDev_RS9110N21_SPI_Unlock_2    (NET_IF                          *p_if);

static void NetDev_RS9110N21_SPI_WrRd_2      (NET_IF                          *p_if,
                                             CPU_INT08U                      *p_buf_wr,
                                             CPU_INT08U                      *p_buf_rd,
                                             CPU_INT16U                       len,
                                             NET_ERR                         *p_err);

static void NetDev_RS9110N21_SPI_ChipSelEn_2 (NET_IF                          *p_if,
                                             NET_ERR                         *p_err);

static void NetDev_RS9110N21_SPI_ChipSelDis_2(NET_IF                          *p_if);

static void NetDev_RS9110N21_SPI_Cfg_2       (NET_IF                          *p_if,
                                             NET_DEV_CFG_SPI_CLK_FREQ         freq,
                                             NET_DEV_CFG_SPI_CLK_POL          pol,
                                             NET_DEV_CFG_SPI_CLK_PHASE        phase,
                                             NET_DEV_CFG_SPI_XFER_UNIT_LEN    xfer_unit_len,
                                             NET_DEV_CFG_SPI_XFER_SHIFT_DIR   xfer_shift_dir,
                                             NET_ERR                         *p_err);
```

Note that since all network device BSP functions are accessed only by function pointer via their corresponding BSP interface structure, they don't need to be globally available and should therefore be declared as `static` .

Also note that although certain device drivers may not need to implement or call all of the above network device BSP function, we recommend that each device's BSP interface structure define all device BSP functions and not assign any of its function pointers to `NULL`. Instead, for any device's unused BSP functions, create empty functions that return `NET_DEV_ERR_NONE`. This way, if the device driver is ever modified to start using a previously unused BSP function, there will at least be an empty function for the BSP function pointer to execute.

Details for these functions may be found in their respective sections in Wireless Device BSP Functions. Templates for network device BSP functions and BSP interface structures can be found in the directory `\Micrium\Software\uC-TCPIP-V2\BSP\Template\`.

Configuring General-Purpose I/O for a Wireless Device

`NetDev_WiFi_CfgGPIO()` configures a specific network device's general-purpose input/ouput (GPIO) on a specific interface. This function is called by a device driver's `NetDev_Init()`.

Each network device's `NetDev_WiFi_CfgGPIO()` should configure all required GPIO pins for the network device. For wireless devices, this function is necessary to configure the power, reset and interrupt pins.

See function `NetDev_WiFi_CfgGPIO` for more information.

Starting a Wireless Device

`NetDev_WiFi_Start()` is used to power up the wireless chip. This function is called by a device driver's `NetDev_WiFi_Start()` each time the interface is started.

Each network device's `NetDev_WiFi_Start()` must set GPIO pins to power up and reset the wireless device. For wireless devices, this function is necessary to configure the power pin and other required pins to power up the wireless chip. Note that a wireless device could require the toggle on the Reset pin to be started or restarted correctly.

See function `NetDev_WiFi_Start` for more information.

Stopping a Wireless Device

`NetDev_WiFi_Stop()` is used to power down a wireless chip. This function is called by a device driver's `NetDev_WiFi_Stop()` each time the interface is stopped.

Each network device's `NetDev_WiFi_Start()` must set GPIO pins to power down the wireless chip to reduce the power consumption. For wireless devices, this function is necessary to configure the power pin and other required pins to power down the wireless chip.

See funciton `NetDev_WiFi_Stop` for more information.

Configuring the Interrupt Controller for a Wireless Device

`NetDev_WiFi_CfgIntCtrl()` is called by a device driver's `NetDev_WiFi_Init()` to configure a specific wireless device's external interrupts for a specific wireless interface.

Each network device's `NetDev_WiFi_CfgIntCtrl()` function must configure without enabling all required interrupt sources for the network device. This means it must configure the interrupt vector address of each corresponding network device BSP interrupt service routine (ISR) handler and disable its corresponding interrupt source. For `NetDev_WiFi_CfgIntCtrl()`, the following actions should be performed:

1. Configure/store each device's network interface number to be available for all necessary `NetDev_WiFi_ISR_Handler()` functions (see section Specifying the Interface Number of

the Device ISR for more information). Even though devices are added dynamically, the device's interface number must be saved in order for each device's ISR handlers to call `NetIF_WiFi_ISR_Handler()` with the device's network interface number.

2. Configure each of the device's interrupts on an interrupt controller (either an external or CPU-integrated interrupt controller). However, vectored interrupt controllers may not require higher-level interrupt controller sources to be explicitly configured and enabled. In this case, you may need to configure the system's interrupt vector table with the name of the ISR handler functions declared in `net_bsp.c`.

`NetDev_WiFi_CfgIntCtrl()` should disable only each devices' interrupt sources. See function `NetDev_WiFi_CfgIntCtrl` for more information.

Enabling and Disabling Wireless Interrupt

Each network device's `NetDev_WiFi_IntCtrl()` function must enable or disable all external required interrupt sources for the wireless device. This means enable or disable its corresponding interrupt source following the enable argument received.

See function `NetDev_WiFi_IntCtrl` for more information.

Configuring the SPI Interface

`NetDev_WiFi_SPI_Init()` initializes a specific network device's SPI controller. This function will be called by a device driver's `NetDev_WiFi_SPI_Init()` when the interface is added.

Each network device's `NetDev_WiFi_SPI_Init()` should configure all required SPI controllers registers for the network device. Since more than one device may share the same SPI bus, this function could be empty if the SPI controller is already configured.

If the SPI bus is not shared with other devices, it is recommended that `NetDev_WiFi_SPI_Init()` configures the SPI controller following the SPI device's communication settings and keep `NetDev_WiFi_SPI_Cfg()` empty.

See `NetDev_WiFi_SPI_Cfg` for more information.

Setting SPI Controller for a Wireless device

`NetDev_WiFi_SPI_Cfg()` configures a specific network device’s SPI communication settings. This function is called by a device driver after the SPI’s bus lock has been acquired and before starting to write and read to the SPI bus.

Each network device’s `NetDev_WiFi_SPI_Cfg()` should configure all required SPI controllers registers for the SPI’s communication setting of the network wireless device. Several aspects of SPI communication may need to be configured, including:

- Clock frequency
- Clock polarity
- Clock phase
- Transfer unit length
- Shift direction

Since more than one device with different SPI communication settings may share the same SPI bus, this function must reconfigure the SPI controller following the device’s SPI communication setting each time the device driver must access the SPI bus. If the SPI bus is not shared with other devices, it’s recommended that `NetDev_SPI_Cfg()` configures the SPI controller following the SPI’s communication setting of the wireless device and to keep this function empty.

See `NetDev_WiFi_SPI_Cfg` for more information.

Locking and Unlocking SPI Bus

`NetDev_WiFi_SPI_Lock()` acquires a specific network device's SPI bus access. This function will be called before the device driver begins to access the SPI. The application should not use the same bus to access another device until the matching call to `NetDev_WiFI_SPI_Unlock()` has been made. If no other SPI device shares the same SPI bus, it's recommended to keep this function empty.

See function `NetDev_WiFi_SPI_Lock` for more information.

Enabling and Disabling SPI Chip select

`NetDev_WiFi_SPI_ChipSelEn()` enables the chip select pin of the wireless device. This function is called before the device driver begins to access the SPI. The chip select pin should stay enabled until the matching call to `NetDev_WiFi_SPI_ChipSelDis()` has been made. The chip select pin is typically “active low.” To enable the device, the chip select pin should be cleared; to disable the device, the chip select pin should be set.

See function `NetDev_WiFi_SPI_ChipSelEn` for more information.

Writing and Reading to the SPI Bus

`NetDev_WiFi_SPI_WrRd()` writes and reads data to and from the SPI bus. This function is called each time the device driver accesses the SPI bus. `NetDev_WiFi_SPI_WrRd()` must not return until the write/read operation is complete. Writing and reading to the SPI bus by using DMA is possible, but the BSP layer must implement a notification mechanism to return from this function only when the write and read operations are entirely completed. See function `NetDev_WiFi_SPI_WrRd` for more information.

Specifying the Interface Number of the Device ISR

`NetDev_ISR_Handler()` handles a network device's interrupts on a specific interface.

Each network device's interrupt, or set of device interrupts, must be handled by a unique BSP-level interrupt service routine (ISR) handler, `NetDev_ISR_Handler()`, which maps each specific device interrupt to its corresponding network interface ISR handler, `NetIF_ISR_Handler()`. For some CPUs, this may be a first- or second-level interrupt handler. The application must configure the interrupt controller to call every network device's unique `NetDev_ISR_Handler()` when the device's interrupt occurs (see `NetDev_CfgIntCtrl`). Every unique `NetDev_ISR_Handler()` must then perform the following actions:

1. Call `NetIF_ISR_Handler()` with the device's unique network interface number and appropriate interrupt type. The network interface number should be available in the device's `NetDev_CfgIntCtrl()` function after configuration (see NetDev_CfgIntCtrl). `NetIF_ISR_Handler()` in turn calls the appropriate device driver's interrupt handler.

   In most cases, each device requires only a single `NetDev_ISR_Handler()`. This is possible when the device's driver is able to determine the device's interrupt type via internal device registers or the interrupt controller. In this case, `NetDev_ISR_Handler()` calls `NetIF_ISR_Handler()` with interrupt type code `NET_DEV_ISR_TYPE_UNKNOWN`.

   However, some devices cannot determine the interrupt type when an interrupt occurs and may therefore require multiple, unique `NetDev_ISR_Handler()`'s, each of which calls `NetIF_ISR_Handler()` with the appropriate interrupt type code.

   Ethernet physical layer (PHY) interrupts should call `NetIF_ISR_Handler()` with interrupt type code `NET_DEV_ISR_TYPE_PHY`.

2. Clear the device's interrupt source, possibly via an external or CPU-integrated interrupt controller source.

See `NetDev_WiFi_ISR_Handler` for more information.

Troubleshooting Guide

This section covers multiple topics to help debug or enhance the performance of a TCP-IP application.

The first three sections introduce common issues you may encounter while using the µC/TCP-IP stack in the initialization process, in your socket application or with your application performance.

The fourth section presents the statistic structures inside the µC/TCPIP stack that could help you debug your application.

The last section shows the µC/TCP-IP stack internal architecture in details.

- Initialization Issues
- Application Issues
- Performance Issues
- Statistics and Error Counters
- Architecture

Initialization Issues

Is your Network Application Compiling?

File Missing Errors

If your compiler return missing file error, refer to section Building and Running the Sample Application to include all µC/TCP-IP stack necessary files to your project and also files from the prerequisite modules. You will also need to add the corresponding paths to your C compiler settings.

Configuration Errors

If you encounter a compiler error of the type '*not #define'd in ...*' or '*NET_CFG_XXX MUST be set to ...*', you have a missing configuration macro or one that is not defined correctly (See section Network Stack Configuration).

Memory Errors

If your compiler returns a memory related error, verify that your MCU memory map is adequate, that your MCU as enough ROM to support the addition of the µC/TCP-IP stack code and also that you have enough RAM to run your Network application. µC/TCP-IP stack can be much RAM-costly depending on the number network buffers your application needs, the number of network interface you have to configured, the network tasks' stack size you configured, etc.

Refer to section LIB Memory Heap Configuration for more details on memory heap allocation necessary for µC/TCP-IP stack.

Refer to section Network Tasks Configuration for more details on the required network tasks configuration.

Port Errors

Is the µC/TCP-IP Stack Initialized Successfully?

RTOS working properly

µC/TCP-IP stack is made to run over a RTOS. Before adding the TCP/IP stack to your project, be sure that your project has correctly initialized the OS, created a starting task and started the OS.

If your OS doesn't tick, the issue is probably with your BSP.

Initialization of prerequisite modules

Your Board Support Package module must be initialize prior to the TCP/IP stack. Refer to function `BSP_Init()` in your BSP files for more details.

The CPU and Memory modules must also be initialize before the TCP/IP stack. Refer to functions `CPU_Init()` and `Mem_Init()` for more details.

Calling Net_Init() Function

1. If the target crashes after calling `Net_Init` , your start task stack size is probably insufficient and must be increase.

2. If the function exits with an error, the problem could be related to the OS configuration or OS objects initialization or Memory configuration (Heap size). The error code return should help you narrowed down the issue. Refer also to section Network Tasks Configuration for more details on the network tasks configuration.

Calling NetIF_Add() Function

If the function doesn't return

1. Verify that your driver Base Address is correct (see section Network Interface Configuration).

2. Could also be a problem with your BSP Clock or GPIO configuration (see section Network Board Support Package).

If the target crashes before returning

1. Verify that your driver Base Address is correct (the target should crash in `NetDev_Init()`if the Base Address is incorrect).

2. Could be a problem in your BSP (the target should crash in `NetDev_Init()` after a BSP call).

3. Your start task stack size could be insufficient.

If the function returns an error

Memory related error

- Increase µC/LIB HEAP size (see section LIB Memory Heap Configuration).

- Or reduce the number of network buffers (see section Network Interface Configuration).

Queue size related error

- Increase Rx and Tx queue size in `net_cfg.h` (see section Network Stack Configuration).

Driver Configuration error

- Invalid configuration, arguments or size in the device configuration (see section Network Interface Configuration).

- PHY mode not supported by the board.

Driver Initialization error

- Invalid configuration, arguments or size in the device configuration (see section Network Interface Configuration).

PHY Invalid Configuration error

- Invalid speed, duplex mode or address (see section Network Interface Configuration).

PHY Initialization error

- If the stack is not able to read and write PHY register, the BSP GPIO configuration must be incorrect (see section Network Board Support Package).

- Bad PHY driver.

Other error

- Refer to `net_err.h` for the error code details.

Calling NetIF_Start() Function

If the function returns an error

- Refer to net_err.h for the error code details.

If the target crashes before returning

- Is the BSP ISR Handler reached? If the ISR Handler crashes before completed: Y our ISR handler could perform an invalid operation. You have a CPU, Compiler or Linker configuration issue. Check interrupt vector table location and interrupt configuration.

Is `Net_DevRx()` reached?

- If reached : Increase the Rx Task stack size.

- If not reached : There is an issue with the OS task switch.

Does it crash when a context switch occurs?

- A probable stack overflow occurs. Increase the network tasks stack sizes (see section Network Tasks Configuration).

Adding a Static Address to the Configured Interface

1. Make sure that the IP address is not used by another host on the network by pinging the address before power up the target.

2. Make sure that the MAC address you configured in your device configuration is not used by another device on the network.
    a. You can use the folloing command on window command line to show the ARP cache: "*arp -a*"

3. If function NetIPv4_CfgAddrAdd or NetIPv6_CfgAddrAdd returned an error refer to the error code and net_err.h for more details on the error type.

Are you able to ping your board?

Once the TCP/IP initialization is done and returned without error, a good first test is to ping your board to validate that the TCP/IP stack is working correctly.

If your target answer back to the ping request, you are ready to add your network application to your project.

In the case that your board does not answer back to ping requests:

1. Be sure that your network setup is correct: The PC used to ping and the target board must be on the same network.

2. Make sure that the ARP request (or NDP in case of IPv6) sent prior to the ICMP echo request is sent out by the PC. Since there is a timeout on the ARP cache, if your PC sends a ARP request that fails to be answered, the PC will assume, for the duration of this timeout, that the remote host is unreachable. Therefore, if you send a ping request to that host during that time, the PC will not send anything and return a fail status. In short, be sure that an ARP (or NDP) request is sent prior to the echo request. Wireshark is a good tool to visually the network traffic on an interface.

3. Is the BSP ISR Handler reached?
    a. In the case it's not reached, there is an issue with your CPU/BSP interrupt configuration.

4. Is `NetDev_Rx()` reached?
    a. In the case the function is not reached:
        i. and the target crashed, try to increase the Rx Task stack size.

        ii. be sure that no task blocks other tasks to run. It could be a OS task switching issue.

    b. In the case the target crashes before reaching `NetDev_Tx()`:
        i. Increase the Rx task stack size.

        ii. Or it maybe an OS task switching issue.

    c. In the case that NetDev_Tx() is not reached but the device does not crashed:
        i. the ARP packet could be discarded because of corruption.

5. If `NetDev_Tx()` reached:
    a. Make sure the data contained in the buffer is not corrupted.

    b. Make sure the data is sent out from the target. If not, it's probably a issue with the network driver.

    c. Make sure `NetIF_DevTxRdySignal()` and `NetIF_TxDeallocTaskPost()` functions are reached.

    d. If you get errors in the driver for Rx or Tx regarding overflow or underflow, verify your BSP clock configuration.

Application Issues

Determine Received UDP Datagram's Interface

If a UDP socket server is bound to the “any” address, then it is not currently possible to know which interface received the UDP datagram. This is a limitation in the BSD socket API. As a solution, the NetSock_CfgIF function has been implemented in the µC/TCP-IP socket API. With this function, it is possible to associate a socket to a specific interface so that all communications with this socket pass through the specified interface.

An other way to guarantee which interface a UDP packet was received on, is to bind the socket server to a specific interface address.

If no interface is linked to a socket (using NetSock_CfgIF) and if a UDP datagram is received on this listening socket bound to the “any” address and the application transmits a response back to the peer using the same socket, then the newly transmitted UDP datagram will be transmitted from the default interface. The default interface may or may not be the interface in which the UDP datagram originated.

Detecting if a Socket is Still Connected to a Peer

Applications may call `NetSock_IsConn()` to determine if a socket is (still) connected to a remote socket (see function NetSock_IsConn()).

Alternatively, applications may make a non-blocking call to `recv()`, `NetSock_RxData()`, or `NetSock_RxDataFrom()` and inspect the return value. If data or a non-fatal, transitory error is returned, then the socket is still connected; otherwise, if ‘0’ or a fatal error is returned, then the socket is disconnected or closed.

Receiving -1 Instead of 0 When Calling recv() for a Closed Socket

When a remote peer closes a socket and the target application calls one of the receive socket functions, µC/TCP-IP will first report that the receive queue is empty and return a -1 for both BSD and µC/TCP-IP socket API functions. The next call to receive will indicate that the socket has been closed by the remote peer.

This is a known issue and will be corrected in subsequent versions of µC/TCP-IP.

The Application Receives Socket Errors Immediately After Reboot

Immediately after a network interface is added, the physical layer device is reset and network interface and device initialization begins. However, it may take up to three seconds for the average Ethernet physical layer device to complete auto-negotiation. During this time, the socket layer will return `NET_SOCK_ERR_LINK_DOWN` for sockets that are bound to the interface in question.

The application should attempt to retry the socket operation with a short delay between attempts until network link has been established.

Please refer to section Network Interface Link State for the detailed information on how to get the link state of an inteface.

Performance Issues

Network and Device Configuration

Number of RX & TX Buffers to Configure

The number of large receive, small transmit and large transmit buffers configured for a specific interface depend on several factors.

1. Desired level of performance.
2. Amount of data to be either transmitted or received.
3. Ability of the target application to either produce or consume transmitted or received data.
4. Average CPU utilization.
5. Average network utilization.
6. Type of connection (UDP or TCP)
7. Number of simultaneous connection.
8. Application/connection priorities

The discussion on the bandwidth-delay product is always valid. In general, the more buffers the better. However, the number of buffers can be tailored based on the application. For example, if an application receives a lot of data but transmits very little, then it may be sufficient to define a number of small transmit buffers for operations such as TCP acknowledgements and allocate the remaining memory to large receive buffers. Similarly, if an application transmits and receives little, then the buffer allocation emphasis should be on defining more transmit buffers. However, there is a caveat:

If the application is written such that the task that consumes receive data runs infrequently or the CPU utilization is high and the receiving application task(s) becomes starved for CPU time, then more receive buffers will be required.

To ensure the highest level of performance possible, it makes sense to define as many buffers as possible and use the interface and pool statistics data in order to refine the number after having run the application for a while. A busy network will require more receive buffers in order to handle the additional broadcast messages that will be received.

In general, at least two large and two small transmit buffers should be configured. This assumes that neither the network or CPU are very busy.

Many applications will receive properly with four or more large receive buffers. However, for TCP applications that move a lot of data between the target and the peer, this number may need to be higher.

Specifying too few transmit or receive buffers may lead to stalls in communication and possibly even dead-lock. Care should be taken when configuring the number of buffers. µC/TCP-IP is often tested with configurations of 10 or more small transmit, large transmit, and large receive buffers.

Number of DMA Descriptors to Configure

If the hardware device is an Ethernet MAC that supports DMA, then the number of configured receive descriptors will play an important role in determining overall performance for the configured interface.

For applications with 10 or less large receive buffers, it is desirable to configure the number of receive descriptors to that of 60% to 70% of the number of configured receive buffers.

In this example, 60% of 10 receive buffers allows for four receive buffers to be available to the stack waiting to be processed by application tasks. While the application is processing data, the hardware may continue to receive additional frames up to the number of configured receive descriptors.

There is, however, a point in which configuring additional receive descriptors no longer greatly impacts performance. For applications with 20 or more buffers, the number of descriptors can be configured to 50% of the number of configured receive buffers. After this point, only the number of buffers remains a significant factor; especially for slower or busy CPUs and networks with higher utilization.

In general, if the CPU is not busy and the µC/TCP-IP Receive task has the opportunity to run often, the ratio of receive descriptors to receive buffers may be reduced further for very high numbers of available receive buffers (e.g., 50 or more).

The number of transmit descriptors should be configured such that it is equal to the number of small plus the number of large transmit buffers.

These numbers only serve as a starting point. The application and the environment that the device will be attached to will ultimately dictate the number of required transmit and receive descriptors necessary for achieving maximum performance.

Specifying too few descriptors can cause communication delays. See Listing F-2 for descriptors configuration.

LF-2(7) Number of receive descriptors. For DMA-based devices, this value is utilized by the device driver during initialization in order to allocate a fixed-size pool of receive descriptors to be used by the device. The number of descriptors *must* be less than the number of configured receive buffers. Micrium recommends setting this value to approximately 60% to 70%f of the number of receive buffers. Non DMA based devices may configure this value to zero.

LF-2(8) Number of transmit descriptors. For DMA-based devices, this value is utilized by the device driver during initialization in order to allocate a fixed-size pool of transmit descriptors to be used by the device. For best performance, the number of transmit descriptors should be equal to the number of small, plus the number of large transmit buffers configured for the device. Non DMA based devices may configure this value to zero.

Configuring Window Sizes

Receive and transmit queue size must be properly configured to optimize performance. It represents the number of bytes that can be queued by one socket. It's important that all socket are not able to queue more data than what the device can hold in its buffers. The size should be also a multiple of the maximum segment size (MSS) to optimize performance. UDP MSS is 1470 and TCP MSS is 1460.

RX and TX maximum queue size is configured using #define in `net_cfg.h`, see Socket Layer Configuration.

RX and TX queue size can be reduce at run time using socket option API (`NetTCP_ConnCfgRxWinSize` and `NetTCP_ConnCfgTxWinSize`).

the following listing shows a calculation example:

```
Number of TCP connection  : 2
    Number of UDP connection  : 0
    Number of RX large buffer : 10
    Number of TX Large buffer : 6
    Number of TX small buffer : 2
    Size of RX large buffer   : 1518
    Size of TX large buffer   : 1518
    Size of TX small buffer   : 60

    TCP MSS RX                = 1460
    TCP MSS TX large buffer   = 1460
    TCP MSS TX small buffer   = 0

    Maximum receive  window   = (10 * 1460)           = 14600 bytes
    Maximum transmit window   = (6  * 1460) + (2 * 0) = 8760  bytes


    RX window size per socket = (14600 / 2) =  7300 bytes
    TX window size per socket = (8760  / 2) =  4380 bytes
```

Reducing the Number of Transitory Errors (NET_ERR_TX)

The number of transmit buffer should be increased. Additionally, it may be helpful to add a short delay between successive calls to socket transmit functions.

Statistics and Error Counters

µC/TCP-IP maintains counters and statistics for a variety of expected or unexpected error conditions. Some of these statistics are optional since they require additional code and memory when enabled, see Network Counters Configuration for further information about how to enable and disabled it.

Statistics

µC/TCP-IP maintains run-time statistics on interfaces and most µC/TCP-IP object pools. If desired, an application can query µC/TCP-IP to find out how many frames have been processed on a particular interface, transmit/receive performance metrics, buffer utilization and more. An application can also reset the statistic pools back to their initial values via appropriate API.

Applications may choose to monitor statistics for various reasons. For example, examining buffer statistics allows you to better manage the memory usage. Typically, more buffers can be allocated than necessary and, by examining buffer usage statistics, adjustments can be made to reduce their number.

Network protocol and interface statistics are kept in an instance of a data structure named `Net_StatCtrs`. This variable may be viewed within a debugger or referenced externally by the application for run-time analysis.

Unlike network protocol statistics, object pool statistics have functions to get a copy of the specified statistic pool and functions for resetting the pools to their default values. These statistics are kept in a data structure called `NET_STAT_POOL` which can be declared by the application and used as a return variable from the statistics API functions.

The data structure is shown below:

```
typedef struct net_stat_pool {
    NET_TYPE           Type;
    NET_STAT_POOL_QTY  EntriesInit;
    NET_STAT_POOL_QTY  EntriesTotal;
    NET_STAT_POOL_QTY  EntriesAvail;
    NET_STAT_POOL_QTY  EntriesUsed;
    NET_STAT_POOL_QTY  EntriesUsedMax;
    NET_STAT_POOL_QTY  EntriesLostCur;
    NET_STAT_POOL_QTY  EntriesLostTotal;
    CPU_INT32U         EntriesAllocatedCtr;
    CPU_INT32U         EntriesDeallocatedCtr;
} NET_STAT_POOL;
```

Listing - Object pool statistics data structure

`NET_STAT_POOL_QTY` is a data type currently set to `CPU_INT16U` and thus contains a maximum count of 65535.

Access to buffer statistics is obtained via interface functions that the application can call (described in the next sections). Most likely, only the following variables in `NET_STAT_POOL` need to be examined, because the `.Type` member is configured at initialization time as `NET_STAT_TYPE_POOL`:

`.EntriesAvail`

This variable indicates how many buffers are available in the pool.

`.EntriesUsed`

This variable indicates how many buffers are currently used by the TCP/IP stack.

`.EntriesUsedMax`

This variable indicates the maximum number of buffers used since it was last reset.

`.EntriesAllocatedCtr`

This variable indicates the total number of times buffers were allocated (i.e., used by the TCP/IP stack).

`.EntriesDeallocatedCtr`

This variable indicates the total number of times buffers were returned back to the buffer pool.

In order to enable run-time statistics, must be enabled, see Network Counters Configuration.

Module pool statistics

You can query the following module to get usage statistics. It can help reduce memory usage and debugging issues regarding resources.

Module	Description	Function API or Variables
ARP	ARP Cache usage	`NetARP_CachePoolStatGet`
IGMP	IPv4 Multicast group statistics	`NetIGMP_HostGrpPoolStat`

NDP	NDP Caches usage	`NetCache_AddrNDP_PoolStat`
		`NetNDP_DestPoolStat`
		`NetNDP_PrefixPoolStat`
		`NetNDP_RouterPoolStat`

Buffer	Interface's buffer usage	`NetBuf_PoolStatGet`
		`NetBuf_RxLargePoolStatGet`
		`NetBuf_TxLargePoolStatGet`
		`NetBuf_TxSmallPoolStatGet`

IP connections	IP connections pool usage	`NetConn_PoolStatGet`
Socket	Sockets usage	`NetSock_PoolStatGet`
TCP	TCP connections usage	`NetTCP_ConnPoolStatGet`
Timer	Timer usage	`NetTmr_PoolStatGet()`

Error Counters

µC/TCP-IP maintains run-time counters for tracking error conditions within the Network Protocol Stack. If desired, the application may view the error counters in order to debug run-time problems such as low memory conditions, slow performance, packet loss, *etc.*

Network protocol error counters are kept in an instance of a data structure named `Net_ErrCtrs`. This variable may be viewed within a debugger or referenced externally by the application for run-time analysis (see `net_ctr.h`).

In order to enable run-time error counters, must be enabled, see Network Counters Configuration.

Architecture

µC/TCP-IP was written to be modular and easy to adapt to a variety of Central Processing Units (CPUs), Real-Time Operating Systems (RTOSs), network devices, and compilers. The figure below shows a simplified block diagram of µC/TCP-IP modules and their relationships.

Notice that all µC/TCP-IP files start with '`net_`'. This convention allows us to quickly identify which files belong to µC/TCP-IP. Also note that all functions and global variables start with '`Net`', and all macros and `#defines` start with '`net_`'.

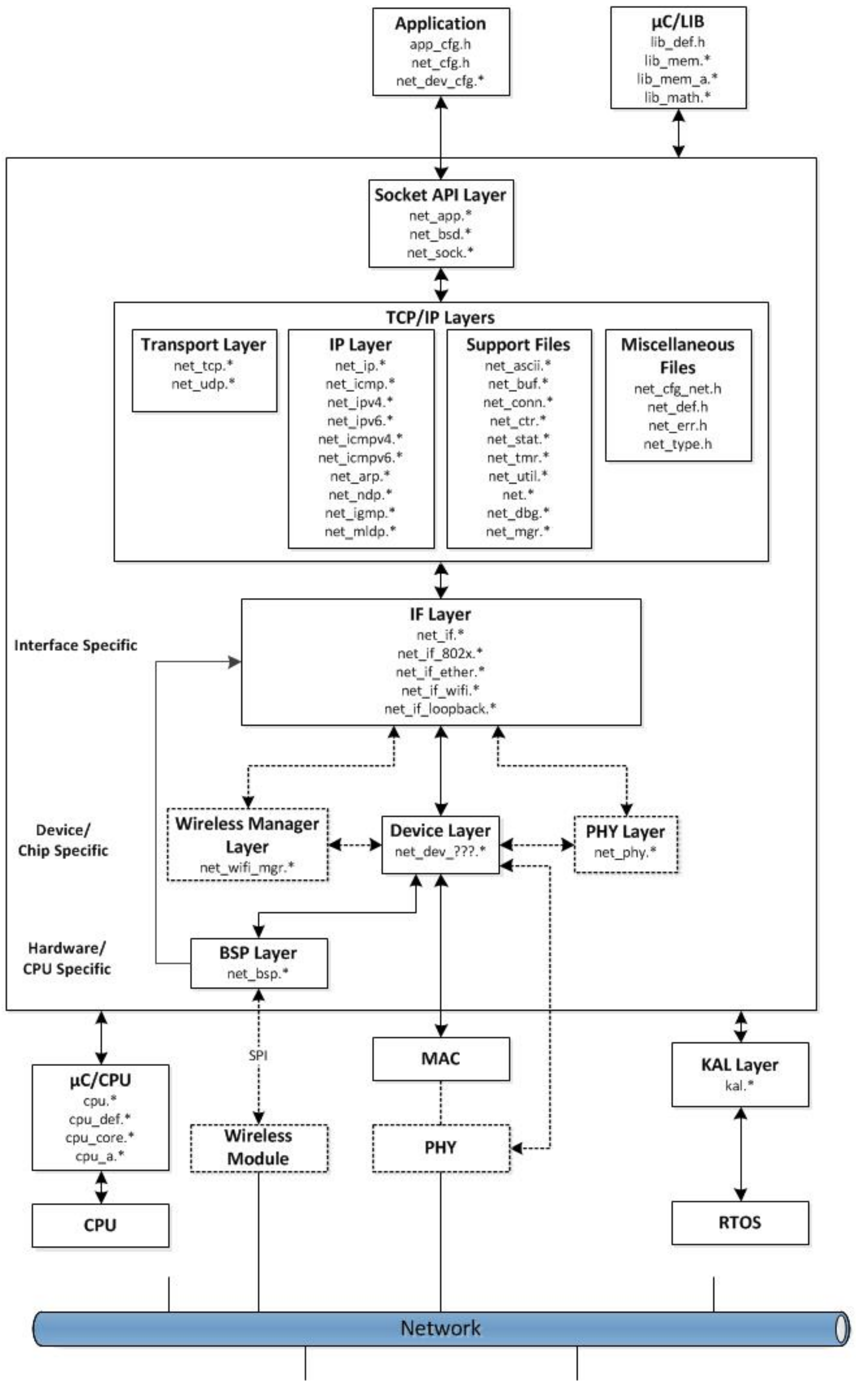

Figure - Module Relationships

Module Relationships

Application

Your application layer needs to provide configuration information to µC/TCP-IP in the form of several C files: `net_dev_cfg.c`, `net_cfg.h`, `net_dev_cfg.c` and `net_dev_cfg.h`:

- Configuration data in `net_cfg.c/h` consists of specifying tasks configuration, the number of timers to allocate to the stack, whether or not statistic counters will be maintained, the number of ARP cache entries, how UDP checksums are computed, and more. One of the most important configurations necessary is the size of the TCP Receive Window. In all, there are approximately 50 `#define` to set. However, most of the `#define` constants can be set to their recommended default value.

- `net_dev_cfg.c` consists of device-specific configuration requirements such as the number of buffers allocated to a device, the MAC address for that device, and necessary physical layer device configuration including physical layer device bus address and link characteristics. Each µC/TCP-IP-compatible device requires that its configuration be specified within `net_dev_cfg.c`.

µC/LIB Libraries

Given that µC/TCP-IP is designed for use in safety critical applications, all “standard” library functions such as `strcpy()`, `memset()`, *etc.* have been rewritten to conform to the same quality as the rest as the protocol stack. All these standard functions are part of a separate Micrium product called µC/LIB. µC/TCP-IP depends on this product. In addition, some data objects in the µC/TCP-IP stack are created at run-time which implies the use of memory allocation from the heap function `Mem_DynPoolAlloc()`.

BSD Socket API Layer

The application will interface with µC/TCP-IP using the BSD socket Application Programming Interface (API). The software developer can either write their own TCP/IP applications using the BSD socket API or, purchase a number of off-the-shelf TCP/IP components (Telnet, Web server, FTP server, etc.),for use with the BSD socket interface. Note that the BSD socket layer is shown as a separate module but is actually part of µC/TCP-IP.

Alternatively, the software developer can use µC/TCP-IP's own socket interface functions (`net_sock.*`). `net_bsd.*` is a layer of software that converts BSD socket calls to µC/TCP-IP socket calls. Of course, a slight performance gain is achieved by interfacing directly to `net_sock.*` functions. Micrium network products use µC/TCP-IP socket interface functions.

TCP/IP Layer

The TCP/IP layer contains most of the CPU, RTOS and compiler-independent code for µC/TCP-IP. There are three categories of files in this section:

1. TCP/IP protocol specific files include:

   a. Generic files:
      ICMP (net_icmp.*)
      IP (net_ip.*)
      TCP (net_tcp.*)
      UDP (net_udp.*)

   b. Files specific to IPv4:
      ARP (`net_arp.*`)
      IPv4 (`net_ipv4.*`)
      ICMPv4 (`net_icmpv4.*`)
      IGMP (`net_igmp.*`)

   c. Files specific to IPv6:
      NDP (`net_ndp.*`)
      IPv6 (`net_ipv6.*`)
      ICMPv6 (`net_icmpv6.*`)
      MLDP (`net_mldp.*`)

2. Support files are:
   ASCII conversions (`net_ascii.*`)
   Buffer management (`net_buf.*`)
   TCP/UDP connection management (`net_conn.*`)
   Counter management (`net_ctr.*`)
   Statistics (`net_stat.*`)
   Timer Management (`net_tmr.*`)
   Other utilities (`net_util.*`).

3. Miscellaneous header files include:
   Master µC/TCP-IP header file (`net.h`)
   File containing error codes (`net_err.h`)
   Miscellaneous µC/TCP-IP data types (`net_type.h`)
   Miscellaneous definitions (`net_def.h`)
   Debug (`net_dbg.h`)
   Configuration definitions (`net_cfg_net.h`)

Network Interface (IF) Layer

The IF Layer involves several types of network interfaces (Ethernet, Token Ring, etc.). However, the current version of µC/TCP-IP only supports Ethernet interfaces, wired and wireless. The IF layer is split into two sub-layers.

`net_if.*` is the interface between higher Network Protocol Suite layers and the link layer protocols. This layer also provides network device management routines to the application.

`net_if_*.*` contains the link layer protocol specifics independent of the actual device (i.e., hardware). In the case of Ethernet, `net_if_ether.*` understands Ethernet frames, MAC addresses, frame de-multiplexing, and so on, but assumes nothing regarding actual Ethernet hardware.

Network Device Driver Layer

As previously stated, µC/TCP-IP works with just nearly any network device. This layer handles the specifics of the hardware, e.g., how to initialize the device, how to enable and disable interrupts from the device, how to find the size of a received packet, how to read a packet out of the frame buffer, and how to write a packet to the device, etc.

In order for device drivers to have independent configuration for clock gating, interrupt controller, and general purpose I/O, an additional file, `net_bsp.c`, encapsulates such details.

`net_bsp.c` contains code for the configuration of clock gating to the device, an internal or external interrupt controller, necessary IO pins, as well as time delays, getting a time stamp from the environment, and so on. This file is assumed to reside in the user application.

Network Physical (PHY) Layer

Often, devices interface to external physical layer devices, which may need to be initialized and controlled. This layer is shown in Figure 2-1 asa “dotted” area indicating that it is not present with all devices. In fact, some devices have PHY control built-in. Micrium provides a generic PHY driver which controls most external (R)MII compliant Ethernet physical layer devices.

Network Wireless Manager

Often, wireless device may need to initialize a command and wait to receive the result (i.e. Scan). This layer manages specific wireless management commands. Micrium provides a generic Wireless Manager which should be able to controls most wireless module.

CPU Layer

µC/TCP-IP can work with either an 8, 16, 32 or even 64-bit CPU, but it must have information about the CPU used. The CPU layer defines such information as the C data type corresponding to 16-bit and 32-bit variables, whether the CPU is little or big endian, and how interrupts are disabled and enabled on the CPU.

CPU-specific files are found in the `...\uC-CPU` directory and are used to adapt µC/TCP-IP to a different CPU, modify either the `cpu*.*` files or, create new ones based on the ones supplied in the uC-CPU directory. In general, it is much easier to modify existing files.

Real-Time Operating System (RTOS) Layer

µC/TCP-IP assumes the presence of an RTOS. An RTOS abstraction layer is also needed allowing µC/TCP-IP to be independent of a specific RTOS (See KAL Layer). µC/TCP-IP consists of three tasks. One task is responsible for handling packet reception, another task for asynchronous transmit buffer de-allocation, and the last task for managing timers. Depending on the configuration, a fourth task may be present to handle loopback operation.

As a minimum, the RTOS:

1. Must be able to create at least three tasks (a Receive task, a Transmit De-allocation task, and a Timer task).

2. Provide semaphore management (or the equivalent) and the μC/TCP-IP needs to be able to create at least two semaphores for each socket and an additional four semaphores for internal use.

3. Provides queuing services.

Kernel Abstraction Layer (KAL)

KAL is a kernel abstraction layer employed by Micrium products to interact with the RTOS used. It can be found in the μC/Common directory. The KAL API is presented in the kal.h file. KAL comes with μC/OS-II and μC/OS-III ports. If a different RTOS is used, a new kal.c file must be develop to match the generic KAL API to the corresponding RTOS functionalities.

Network Buffer Architecture

μC/TCP-IP uses both small and large network buffers:

- Network buffers
- Small transmit buffers
- Large transmit buffers
- Large receive buffers

A single network buffer is allocated for each small transmit, large transmit and large receive buffer. Network buffers contain the control information for the network packet data in the network buffer data area. Currently, network buffers consume approximately 200 bytes each. The network buffers' data areas are used to buffer the actual transmit and receive packet data. Each network buffer is connected to the data area via a pointer to the network buffer data area, and both move through the network protocol stack layers as a single entity. When the data area is no longer required, both the network buffer and the data area are freed. The figure below depicts the network buffer and data area objects.

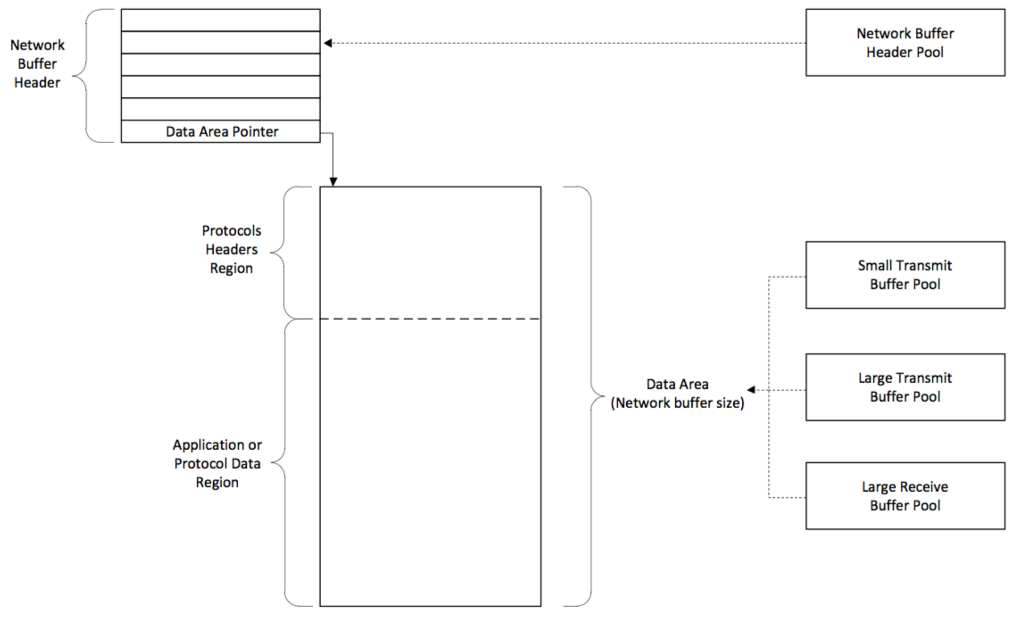
Network Buffer Header
Data Area Pointer
Network Buffer Header Pool
Protocols Headers Region
Application or Protocol Data Region
Data Area (Network buffer size)
Small Transmit Buffer Pool
Large Transmit Buffer Pool
Large Receive Buffer Pool

Figure - Network Buffer Architecture

Network Buffer Sizes

The following table shows how each network buffer should be configured to handle the majority of worst cases.

Type of network buffer	Size
Receive Large Buffer	1518 + Alignment
Transmit Large Buffer	1518 + Alignment
Transmit Small Buffer	64 + Alignment

TCPIP Tasks and Priorities

The user application interfaces to µC/TCP-IP via a well known API called BSD sockets (or µC/TCP-IP's internal socket interface). The application can send and receive data to/from other hosts on the network via this interface.

The BSD socket API interfaces to internal structures and variables (i.e., data) that are maintained by µC/TCP-IP. A binary semaphore (the global lock in the figure µC/TCP-IP Task model) is used to guard access to this data to ensure exclusive access. In order to read or write to this data, a task needs to acquire the binary semaphore before it can access the data and release it when finished. Of course, the application tasks do not have to know anything about this semaphore nor the data since its use is encapsulated by functions within µC/TCP-IP.

The figure µC/TCP-IP Task model shows a simplified task model of µC/TCP-IP along with application tasks.

µC/TCP-IP defines three internal tasks: a Receive task, a Transmit De-allocation task, and a Timer task. The Receive task is responsible for processing received packets from all devices. The Transmit De-allocation task frees transmit buffer resources when they are no longer required. The Timer task is responsible for handling all timeouts related to TCP/IP protocols and network interface management.

When setting up task priorities, we generally recommend that tasks that use µC/TCP-IP's services be configured with higher priorities than µC/TCP-IP's internal tasks. However, application tasks that use µC/TCP-IP should voluntarily relinquish the CPU on a regular basis. For example, they can delay or suspend the tasks or wait on µC/TCP-IP services. This is to reduce starvation issues when an application task sends a substantial amount of data.

We recommend that you configure the network interface Transmit De-allocation task with a higher priority than all application tasks that use µC/TCP-IP network services; but configure the Timer task and network interface Receive task with lower priorities than almost other application tasks.

See also Operating System Configuration.

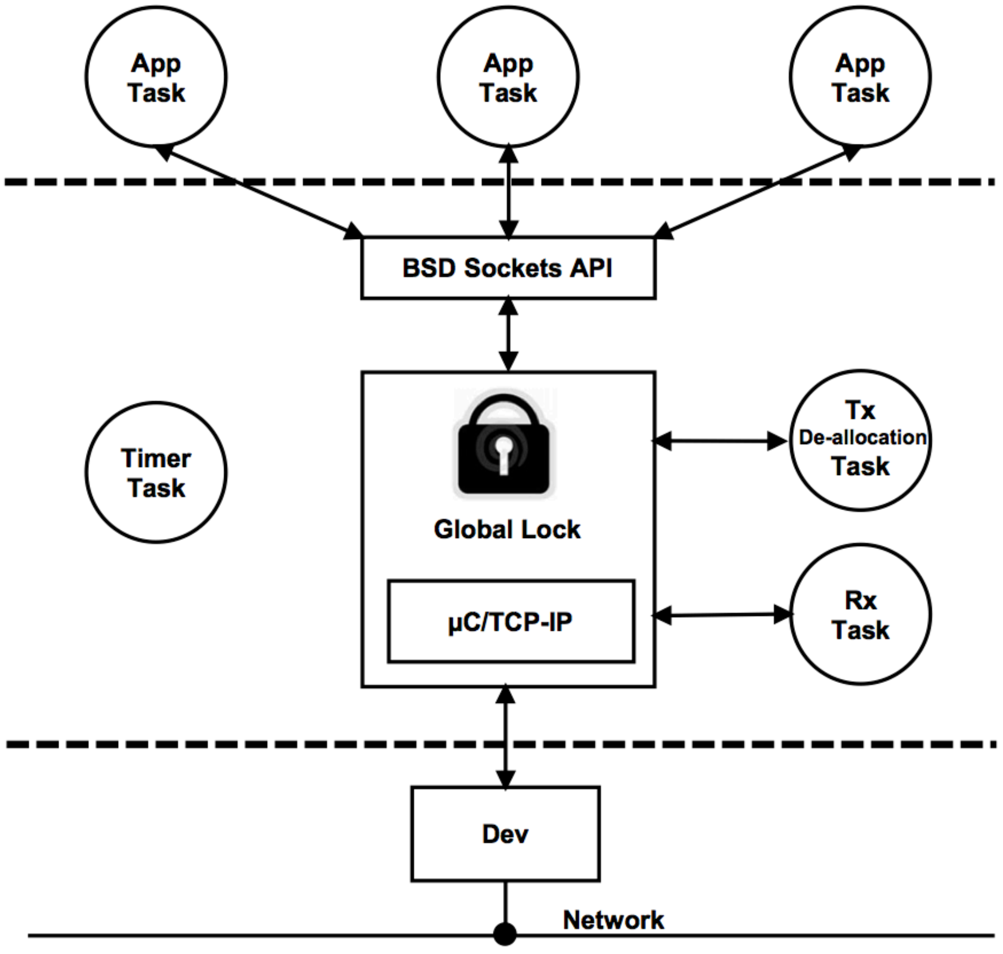

Figure - µC/TCP-IP Task model

Receiving a Packet

This figure shows a simplified task model of μC/TCP-IP when packets are received from the device.

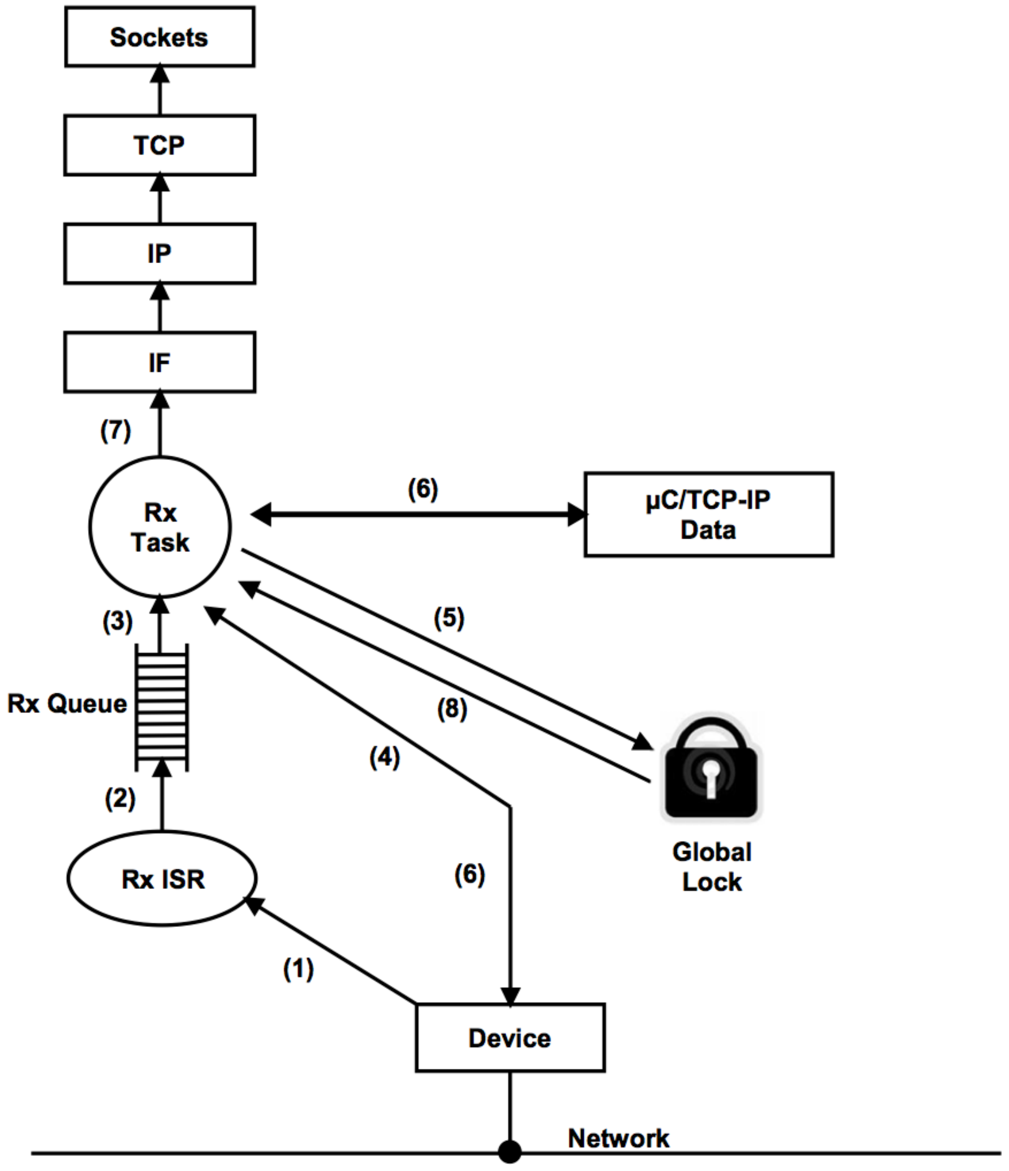

Figure - μC/TCP-IP Receiving a Packet

1. A packet is sent on the network and the device recognizes its address as the destination for the packet. The device then generates an interrupt and the BSP global ISR handler is called for non-vectored interrupt controllers. Either the global ISR handler or the vectored interrupt controller calls the Net BSP device specific ISR handler, which in

turn indirectly calls the device ISR handler using a predefined Net IF function call. The device ISR handler determines that the interrupt comes from a packet reception (as opposed to the completion of a transmission).

2. Instead of processing the received packet directly from the ISR, it was decided to pass the responsibility to a task. The Rx ISR therefore simply “signals” the Receive task by posting the interface number to the Receive task queue. Note that further Rx interrupts are generally disabled while processing the interrupt within the device ISR handler.

3. The Receive task does nothing until a signal is received from the *Rx ISR*.

4. When a signal is received from an Ethernet device, the Receive task wakes up and extracts the packet from the hardware and places it in a receive buffer. For DMA based devices, the receive descriptor buffer pointer is updated to point to a new data area and the pointer to the receive packet is passed to higher layers for processing.

5. µC/TCP-IP maintains three types of device buffers: small transmit, large transmit, and large receive. For a common Ethernet configuration, a small transmit buffer typically holds up to 256 bytes of data, a large transmit buffer up to 1500 bytes of data, and a large receive buffer 1500 bytes of data. Note that the large transmit buffer size is generally specified within the device configuration as 1594 or 1614 bytes (see Chapter 9, “Buffer Management” for a precise definition). The additional space is used to hold additional protocol header data. These sizes as well as the quantity of these buffers are configurable for each interface during either compile time or run time.

   Buffers are shared resources and any access to those or any other µC/TCP-IP data structures is guarded by the binary semaphore that guards the data. This means that the Receive task will need to acquire the semaphore before it can receive a buffer from the pool.

6. The Receive task gets a buffer from the buffer pool. The packet is removed from the device and placed in the buffer for further processing. For DMA, the acquired buffer pointer replaces the descriptor buffer pointer that received the current frame. The pointer to the received frame is passed to higher layers for further processing.

7. The Receive task examines received data via the appropriate link layer protocol and determines whether the packet is destined for the ARP or IP layer, and passes the buffer to the appropriate layer for further processing. Note that the Receive task brings the data

all the way up to the application layer and therefore the appropriate μC/TCP-IP functions operate within the context of the Receive task.

8. When the packet is processed, the lock is released and the Receive task waits for the next packet to be received.

Transmitting a Packet

This figure shows a simplified task model of µC/TCP-IP when packets are transmitted through the device.

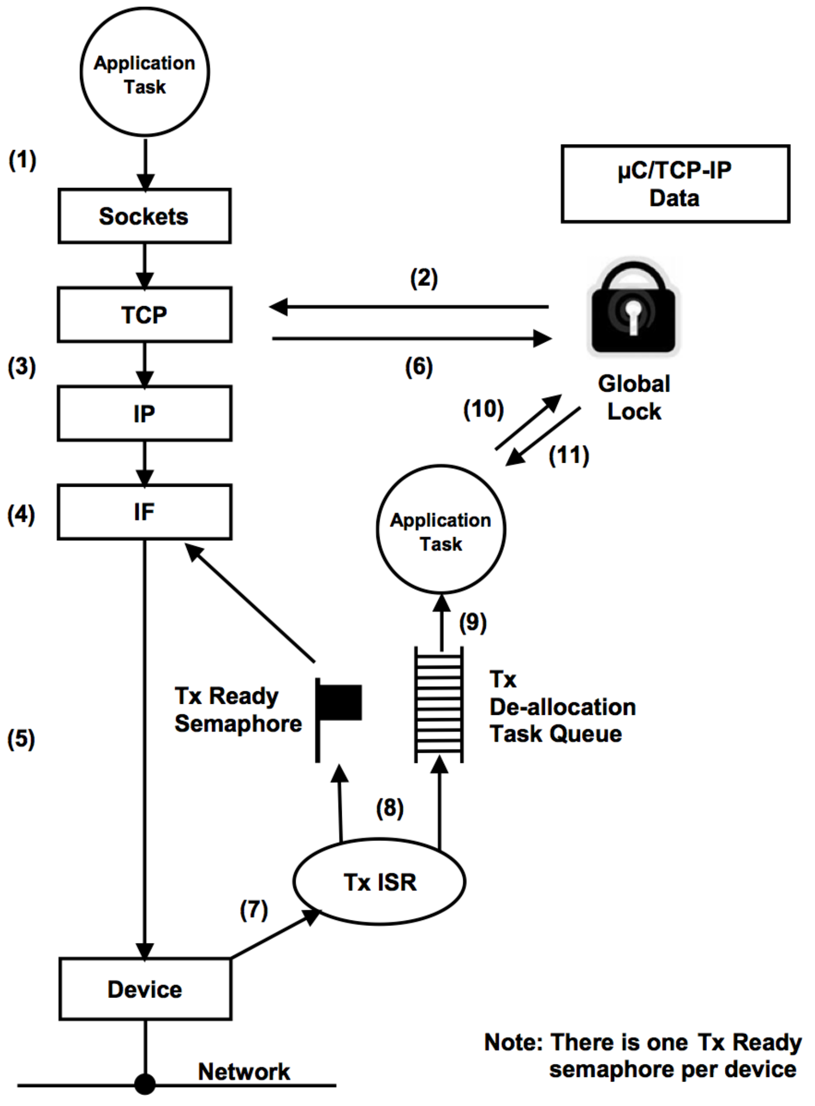

Figure - µC/TCP-IP Sending a Packet

1. A task (assuming an application task) that wants to send data interfaces to µC/TCP-IP through the BSD socket API.

2. A function within µC/TCP-IP acquires the binary semaphore (i.e., the global lock) in order to place the data to send into µC/TCP-IP's data structures.

3. The appropriate µC/TCP-IP layer processes the data, preparing it for transmission.

4. The task (via the IF layer) then waits on a counting semaphore, which is used to indicate that the transmitter in the device is available to send a packet. If the device is not able to send the packet, the task blocks until the semaphore is signaled by the device. Note that during device initialization, the semaphore is initialized with a value corresponding to the number of packets that can be sent at one time through the device. If the device has sufficient buffer space to be able to queue up four packets, then the counting semaphore is initialized with a count of 4. For DMA-based devices, the value of the semaphore is initialized to the number of available transmit descriptors.

5. When the device is ready, the driver either copies the data to the device internal memory space or configures the DMA transmit descriptor. When the device is fully configured, the device driver issues a transmit command.

6. After placing the packet into the device, the task releases the global data lock and continues execution.

7. When the device finishes sending the data, the device generates an interrupt.

8. The Tx ISR signals the Tx Available semaphore indicating that the device is able to send another packet. Additionally, the Tx ISR handler passes the address of the buffer that completed transmission to the Transmit De-allocation task via a queue which is encapsulated by an OS port function call.

9. The Transmit De-allocation task wakes up when a device driver posts a transmit buffer address to its queue.

10. The global data lock is acquired. If the global data lock is held by another task, the Transmit De-allocation task must wait to acquire the global data lock. Since it is recommended that the Transmit De-allocation task be configured as the highest priority µC/TCP-IP task, it will run following the release of the global data lock, assuming the queue has at least one entry present.

11. The lock is released when transmit buffer de-allocation is finished. Further transmission and reception of additional data by application and µC/TCP-IP tasks may resume.

Timer Management

µC/TCP-IP manages software timers used to keep track of various network-related timeouts. Timer management functions are found in `net_tmr.*`. Timers are required for:

Network interface/device driver link-layer monitor	1 total
Network interface performance statistics	1 total
ARP cache management	1 per ARP cache entry
IP fragment reassembly	1 per fragment chain
Various TCP connection timeouts	Up to 7 per TCP connection
Performance monitor task	1 total

Of the three mandatory µC/TCP-IP tasks, one of them, the timer task, is used to manage and update timers. The timer task updates timers periodically. `NET_TMR_CFG_TASK_FREQ` determines how often (in Hz) network timers are to be updated. This value *must not* be configured as a floating-point number. This value is typically set to **10 Hz**.

`NET_TMR_CFG_NBR_TMR` determines the number of timers that µC/TCP-IP will be managing. Of course, the number of timers affect the amount of RAM required by µC/TCP-IP. Each timer requires 12 bytes plus 4 pointers.

It is recommended to set `NET_TMR_CFG_NBR_TMR` with at least **`12`** timers, but a better starting point may be to allocate the maximum number of timers for all resources.

For instance, if 10 ARP caches are configured (`NET_ARP_CFG_CACHE_NBR = 10`), 10 NDP caches are configured (`NET_NDP_CFG_CACHE_NBR= 10`) and 10 TCP connections are configured (`NET_TCP_CFG_NBR_CONN = 10`); the maximum number of timers for these resources is 1 for the Network Performance Monitor, 1 for the Link State Handler, (`10 * 1`) for the ARP caches, (`10 * 1`) for the NDP caches and (`10 * 7`) for TCP connections:

```
# Timers = 1 + 1 + (10 * 1) + (10 * 1) + (10 * 7) = 92
```

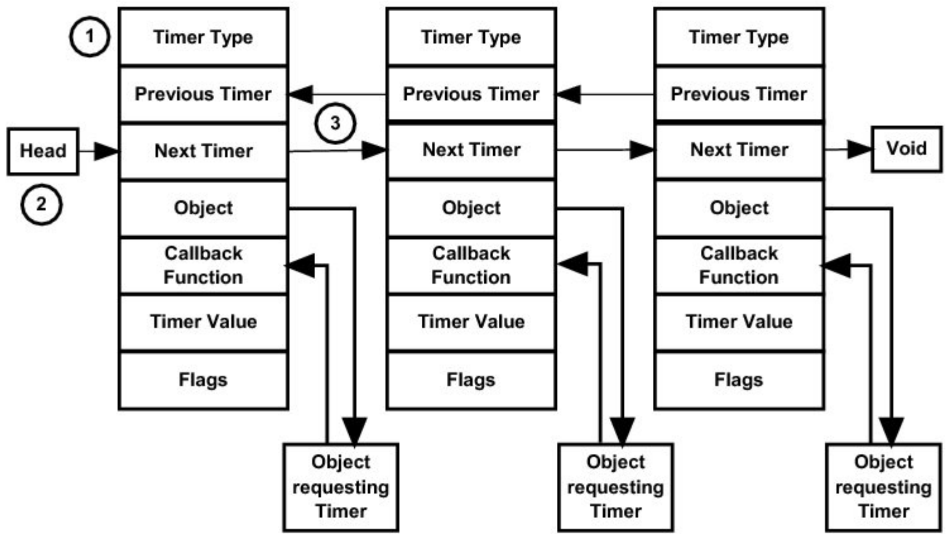

Figure - Timer List

1. Timer types are either `NONE` or `TMR` , meaning unused or used. This field is defined as ASCII representations of network timer types. Memory displays of network timers will display the timer TYPEs with their chosen ASCII name.

2. To manage the timers, the head of the timer list is identified by `NetTmr_TaskListHead`, a pointer to the head of the Timer List.

3. `PrevPtr` and `NextPtr` doubly link each timer to form the Timer List.

The flags field is currently unused.

Network timers are managed by the Timer task in a doubly-linked Timer List. The function that executes these operation is the `NetTmr_TaskHandler()` function. This function is an operating system (OS) function and *should* be called only by appropriate network-operating system port function(s). `NetTmr_TaskHandler()` is blocked until network initialization completes.

`NetTmr_TaskHandler()` handles the network timers in the Timer List by acquiring the global network lock first. This function blocks all other network protocol tasks by pending on and acquiring the global network lock. Then it handles every network timer in Timer List by decrementing the network timer(s) and for any timer that expires, execute the timer's callback function and free the timer from Timer List. When a network timer expires, the timer is be freed *prior* to executing the timer callback function. This ensures that at least one timer is available if the timer callback function requires a timer. Finally, `NetTmr_TaskHandler()` releases the global network lock.

New timers are added at the head of the Timer List. As timers are added into the list, older timers migrate to the tail of the Timer List. Once a timer expires or is discarded, it is removed.

`NetTmr_TaskHandler()` handles of all the valid timers in the Timer List, up to the first corrupted timer. If a corrupted timer is detected, the timer is discarded/unlinked from the List. Consequently, any remaining valid timers are unlinked from Timer List and are not handled. Finally, the Timer task is aborted.

Since `NetTmr_TaskHandler()` is asynchronous to ANY timer Get/Set, one additional tick is added to each timer's count-down so that the requested timeout is *always* satisfied. This additional tick is added by NOT checking for zero ticks after decrementing; any timer that expires is recognized at the next tick.

A timer value of 0 ticks/seconds is allowed. The next tick will expire the timer.

The `NetTmr_***()` functions are internal functions and should not be called by application functions. This is the reason they are not described here or in TCPIP API Reference Core. For more details on these functions, please refer to the `net_tmr.*` files.