PicoTCP User Documentation

April 26, 2013

April 26, 2013

Contents

1. Overview

PicoTCP is a complete TCP/IP stack, intended for embedded devices and designed to run on different architectures and networking hardware. The architecture of the stack allows easy selection of the features needed for any particular use, taking into account the sizing and the performance of the platform on which the code is to run. Even if it is designed to allow for size and performance constraints, the chosen approach is to comply with the latest standards in the telecommunications research, including the latest proposals, in order to achieve the highest standards for today's inter-networking communications. PicoTCP is distributed as a library to be integrated with application and form a combination for any hardware-specific firmware.

The main characteristics of the library are the following:

- **Modularity** Each component of the stack is deployed in a separate module, allowing the selection at compile time of the components needed to be included for any specific platform, depending on the particular use case. We know that saving memory and resources is often mission-critical for a project, and therefore PicoTCP is fully focussed on saving up to the last byte of memory.
- **Code Quality** Every component added to the stack must pass a complete set of validation tests. Before new code can be introduced it is scanned and proof-checked by three separate levels of quality enforcement. The process related to the validation of the code is one of the major tasks of the engineering team. In the top-down approach of the design, a new module has to pass the review of our senior architects, to have it comply with the general guidelines. The development of the smaller components is done in a test-driven way, providing a specific unit test for each function call. Finally, functional non-regression tests are performed after the feature development is complete, and all the tests are automatically scheduled to run several times per day to check for functional regressions.
- **Adherence to the standards** The protocols included in the stack are done following stepare designed by following meticulously the guidelines provided by the International Engineering Task Force (IETF) with regards to inter-networking communication. A strong adherence to the standards guarantees a smooth integration with all the existing TCP/IP stacks, when communicating with both other embedded devices and with the PC/server world.
- **Features** A fully-featured protocol implementation including all those non-mandatory features means better data-transfer performances, coverage of rare/unique network scenarios and topologies and a better integration with all types of networking hardware devices.
- **Transparency** The availability of the source code to the Free Software community is an important added value of PicoTCP. The constant peer reviews and constructive comments on the design and the development choices that PicoTCP receives from the academic world and from several hundreds of hobbyists and professionals who read the code, are an essential element in the quality build-up of the product.
- **Simplicity** The APIs provided to access the library facilities, both from the applications as well as from the device drivers, are small and well documented. This concurs with the

goal of the library to facilitate the integration with the surroundings and minimize the time used to combine the stack with existing code. The support required to port to a new architecture is so small it is reduced to a set of macros defined in a header file specific for the platform.

2. Usage and platform integration

2.1 Requirements and Configuration

PicoTCP is designed to be portable and versatile. Modules can be activated at compile-time, or excluded from the compilation in order to reduce the build size or save resources at runtime. This characteristic allows an embedded application to create different types of appliances, starting from a small forwarding multi-protocol switch, to fully-featured TCP hosts, supporting internal applets as well as generic POSIX-compliant socket interfaces.

2.2 Supported features

- **Device layer** Facilities for device driver are offered in a simple structure and API.
- **ARP** The stack can use the "Address Resolution Protocol" to retrieve the MAC addresses of other hosts in the network.
- **IPv4** The network layer supports the IPv4 network layer protocol. An API is provided in order to access all the addressing and routing related functionalities.
- **ICMP** Also the "Internet Control Message Protocol" is implemented. This protocol provides the system to send error messages, do a ping, ...
- **NAT** The stack supports "Network Address Translation" to hide addresses from internal networks to the outside. The API also supports functions for port forwarding.
- **multicast sockets** The stack supports multicast (one-to-many) sockets and addresses in order to send and receive data to/from multicast groups.
- **IGMP** As an integration for the multicast features above, IGMP version 2 is supported to manage the membership to multicast groups.
- **UDP** The stack can use the "User Datagram Protocol" as a transport protocol for connection-less communication between sockets.
- **TCP** The stack supports the connection-oriented "Transport Control Protocol" for reliable communications. The TCP implementation is fully featured and the most commonly used extensions are included.
- **Sockets** The user applications on different host use the socket API to communicate. The socket API is based on the latest POSIX (1-2008) specifications, while not being fully compliant due to the fact that it is designed to run in a single threading unit. Blocking functionalities are reproduced via callback triggering as described in the socket API documentation.
- **DNS client** A small DNS client is provided to resolve an IP address for a given name. The API supports setting several DNS servers and a small cache.

- **DHCP client** A DHCP client can request an IP lease from a DHCP server to set the IP adress of the device.
- **DHCP server** Also a small DHCP server is included to hand out IP addresses to hosts in the network.
- **Linux development and test facilities** The stack is developed entirely on a Linux system. Several tools are easily available and/or included to develop and test user applications. (tun/tap devices, vde, tcp benchmark test, ...)

2.3 Enabling modules

Each module, option and feature included in the code base must be explicitly enabled by defining a specific PICO_SUPPORT_ preprocessor variable. If the default Makefile is used to compile PicoTCP, this can be done using command line options when running make. The syntax required to compile the protocol in a library (the default Makefile target) is the following:

```
make [MAKE_ARG=VALUE] [...]
```

2.3.1 Compile-time options

A few compile-time options can be specified using the command line arguments of make to modify the result of the build. Global options that affect the build are the following:

Argument	Possible values	Default value	Description
DEBUG	0,1	1	When enabled (=1), the resulting library will contain debug symbols. The size of the library will be much larger than the production build, but it will be possible to run the stack into a debugger to inspect its behaviour. When the option is disabled (=0), the library will be optimized for size in flash, resulting in a smaller binary to be used in production.
PREFIX	any valid path	./build	The target directory where the library and all the objects will be placed after the compilation.
ENDIAN	little, big	little	Force to build against little-endian or big-endian architecture.
CROSS_COMPILE	compiler prefix		Use a cross compile prefix when calling the binaries needed to build.

TCP	0,1	1	Enables the support for Transmission Control Protocol by allowing the usage of stream sockets.
UDP	0,1	1	Enables the support for User Datagram Protocol by allowing the usage of datagram sockets.
IPV4	0,1	1	Enables the support for basic IP networking functionalities. At least one network protocol is required for most of the features to work, as all types of sockets depend on the networking layer.
NAT	0,1	1	Activates the support for network address translation to IPv4.
ICMP4	0,1	1	Enables the support for control messages over IPv4, (not including the ping functionalities).
MCAST	0,1	1	If enabled, the support for multicast sockets will be included in the resulting library.
DEVLOOP	0,1	1	If enabled, a loopback device will be added to the stack, and can be configured to run local traffic.
PING	0,1	1	When activated, the ping API will be available to test whether the hosts on the network are reachable. Requires ICMP4 support.
DNS_CLIENT	0,1	1	This feature is required to resolve host names into IP addresses and vice-versa.
DHCP_CLIENT	0,1	1	When activated, it will be possible to get the IP address for network devices automatically, when a DHCP server is present on the network.
DHCP_SERVER	0,1	1	If activated, it will be possible to run a small DHCP server to provide addresses for automatic configuration to the other hosts in the network.

2.3.2 Architecture support

By default, the stack will be compiled to run in a process on a POSIX system, e.g. to be linked to a Linux application. To change this behavior and produce a library linked to a specific board-support package (BSP) among those supported, it is sufficient to set the command line argument variable ARCH to a specific value. The architectures supported by the stack are the following:

ARCH keyword	CPU	Reference hardware
stm32	ARM Cortex M4-F	ST Microelectronics evaluation board "STM32f4 Discovery"
stellaris	ARM Cortex LM3S-6965	Texas Instrument Evaluation Kit "Codesourcery LM3S6965 ETH"

2.4 Target requirements

PicoTCP can run on several different hardware architectures and can be integrated with virtually any operating system or within a standalone application. It is possible to run PicoTCP on big-endian as well as little-endian CPU configurations. PicoTCP uses gcc-specific tags that may not be compatible with other compilers. The amount of resources needed may vary depending on the modules that are compiled-in. However, adapting to a specific hardware platform or for a particular use may require some integration effort.

2.4.1 Porting PicoTCP to a target system

Warning: ensure that the Board Support Package provided by your hardware supplier is distributed under the terms of a license compatible with the PicoTCP license, described in the Appendix of this document.

PicoTCP relies on a simple set of system-specific calls that must be implemented externally from the target. Briefly, the interface needed for the stack to run is composed by:

- A mechanism to allocate dynamic memory on the system
- A stable time-source to update its internal counters

For the memory allocation interface, two symbols have to be defined by the system:

```
void *pico_zalloc(int size) - (memory allocation)
void pico_free(void *ptr) - (memory release)
```

- `pico_zalloc` Must allocate an object of the given size size in memory and set the content of the allocated memory to zero. A pointer to the address 0 will indicate an allocation failure.
- `pico_free` Must release the memory assigned to the object previously allocated at the address ptr.

For the time keeping, the following objects must be defined by the system:

- `static inline unsigned long PICO_TIME(void)`
  Returns current time expressed in seconds
- `static inline unsigned long PICO_TIME_MS(void)`
  Returns current time expressed in milliseconds
- `static inline void PICO_IDLE(void)`
  Sleep between two consecutive iterations inside the main protocol loop (e.g. to yield the CPU to some other functionality on the sytem)

As an alternative to defining the time-keeping procedure in the asynchronous functions `PICO_TIME()` and `PICO_TIME_MS()`, it is possible to use an interrupt handler linked to a fixed interval time source, increasing the volatile global variable `pico_tick`. If done this way, the two functions may return the values of `(pico_tick / 1000)` and `pico_tick`, respectively.

Finally, whenever debug information is needed, the system will have to provide a `dbg()` function that accepts the same variadic arguments model as a standard `printf()`.

2.4.2 Defining a new architecture support

If all the above requirements are satisfied, PicoTCP expects those functions to be mapped to existing code in the BSP of the architecture. An easy way to do so is by means of a new architecture-specific header file under the `include/arch` subdirectory. Since all the functions above must already be implemented outside the PicoTCP tree, the library will have to be linked to the system support library, either during compilation or at at a subsequent stage when the resulting firmware is being generated. For this reason, a prototype of all the functions used to implement the functionalities requested by the BSP must be included from the architecture support header file or incorporated into the file itself.

For instance, if the BSP for an architecture called "foobar" provides the following functions:

```
void *custom_allocate_and_zero(int size);
void *custom_free(void *mem);
int print_serial_debug(...);
```

and an interrupt handler is attached to a time source in order to increment the `pico_tick` variable every millisecond, a possible architecture-specific file (under `arch/pico_foobar.h`) should look like the following:

```
/* repeat the prototypes used */
extern void *custom_allocate_and_zero(int size);
extern void *custom_free(void *mem);
extern int print_serial_debug(...);

#define dbg print_serial_debug
#define pico_zalloc(x) custom_allocate_and_zero(x)
#define pico_free(x) custom_free(x)

static inline unsigned long PICO_TIME(void)
```

```
{
  return pico_tick / 1000;
}

static inline unsigned long PICO_TIME_MS(void)
{
  return pico_tick;
}

static inline void PICO_IDLE(void)
{
  unsigned long tick_now = pico_tick;
  while(tick_now == pico_tick);
}
```

Once the architecture-specific file is created, it is time to add the architecture-specific support to the `pico_config.h` file, the same way it is done for the existing architectures, using an additional preprocessor elif block:

```
#elif defined FOOBAR
#include "arch/pico_foobar.h"
```

From this point on, it is sufficient to define a preprocessor variable with the keyword chosen for the architecture, all in capitals (`FOOBAR` in this example case). The final step is to create a block in the main PicoTCP makefile that also sets the compiler flags needed to produce objects that are compatible with and/or optimized for the foobar architecture. Additionally, this block also contains the definition of the keyword preprocessor macro in order to have the correct arch-specific header included:

```
ifeq ($(ARCH),foobar)
  CFLAGS+=-mcustom-foobar-code -DFOOBAR
endif
```

To compile for the foobar architecture, it is now sufficient to run

```
make ARCH=foobar
```

2.5 Network devices integration

Every device driver must define its own interface to communicate with the stack. This interface is accessed via the `pico_device` structure. Every device implements an instance of this structure by populating the following mandatory fields:

- **overhead** - A positive integer indicating the amount of bytes required by the device driver to implement its header. This is used whenever a network layer allocates a new packet to be sent through this device. If a value is specified here, it will be possible for the

device to seek back in the frame scheduled for sending, and subsequently copy any header information in front of it. Devices dealing with pure stack frames or subparts of it (e.g. Ethernet) should have overhead set to 0.

- The callback **send** - must be a pointer to a function internally defined in the device driver module. This function will be called every time a frame must be injected in the network. The module can implement a generic **send** function for all the registered devices, as the device field will be passed as the first argument. The callback prototype is the following:

  ```
  int (*send)(struct pico_device *self, void *buf, int len);
  ```

  If the device can immediately inject the frame at address `buf` of length `len`, it returns back to the caller the length of the frame injected. If the device is currently busy, this function can safely return 0, and the stack will retry the same operation again later.

- The callback **poll** - must be a pointer to a function internally defined in the device driver module. This function will be called periodically by the stack, to request a synchronization on the incoming frames. The prototype is the following:

  ```
  int (*poll)(struct pico_device *self, int loop_score);
  ```

  The poll function must check if the device is ready to receive frames, and for each frame that is directed to the stack, it will call the library function `pico_stack_recv()`. This function will deliver the received frame to the stack.

  The `loop_score` variable represents the maximum amount of frames that the stack can process during this call, i.e. the maximum amount of calls to `pico_stack_recv()` that can be performed during this iterations. The device driver should loop around the packet delivery operation and decrease the loop_score by one every time a frame is delivered to the stack. If during the iteration all the score was used, poll will return 0.

  NOTE: The poll function must return **immediately** and must never block on hardware-specific operations. If the device is interrupt-driven, the integration will have to provide a mechanism to defer the reception until the next call back to poll. Calling `pico_stack_recv()` is only allowed from inside the `poll()` callback, thus a two-halves interface interrupt management design is required, and any memory structure shared between the two halves must be protected against concurrent access accordingly.

- The callback **destroy** - a pointer to a function that deallocates the device structure itself and frees all the structures that were possibly allocated by the driver during device creation.

A device driver will have a simple two-functions library API exported in a header file using the same name, in the modules directory. The two functions to export will be:

- A **create** function, accepting any argument required for the internal device configuration, that returns a pointer to the newly allocated device. The function must allocate the device and finally call the library function `pico_device_init()` in order to register the device into the stack. The `pico_device_init()` function accepts the following arguments:
  - the device allocated just before
  - a null-terminated string containing a unique device name for the device to be inserted in the system (e.g. "eth0")

  - a pointer to an Ethernet address in the form of a previously allocated `pico_ethdev` structure, containing the hardware address to be used by the stack for datalink addressing. If no hardware-specific address is provided to `pico_device_init()` is provided (i.e. a `NULL` pointer is passed), the newly created device will be directly attached to the network layer and it will have to provide and process valid IP packets without further encapsulation.

- A destroy routine, accepting the previously allocated device pointer to free all the associated structures.

The way to expand the device driver interface is by simply creating a new specific structure that contains it and thus inherits all the capabilities of the standard structure but also holds the required hardware-specific information. The three callbacks will always receive a pointer to the beginning of the `pico_device` structure, but the memory area that follows the structure can be used to keep track of the device hardware-specific context.

Naming conventions must be followed for the two functions exposed to the user interface to create and destroy the device. The functions must be named `pico_X_create()` and `pico_X_destroy()`, where X is the unique name of the device driver.

As an example of a very simple device driver, directly attached to the networking layer using the valid naming convention for the **send/poll/create/destroy** interfaces are contained in the source file `modules/pico_dev_null.c` and its header `modules/pico_dev_null.h`.

3. API Documentation

The following sections will describe the API for picoTCP.

3.1 IPv4 functions

3.1.1 pico_ipv4_to_string

Description

Convert the internet host address IP to a string in IPv4 dotted-decimal notation. The result is stored in the char array that ipbuf points to. Little endian or big endian is not taken into account. The worst case memory requirement for ipbuf is 16 bytes (12 digits, 3 periods and '\0'). For example: 0xC0A80101 becomes 192.168.1.1

Function prototype

```
int pico_ipv4_to_string(char *ipbuf, const uint32_t ip);
```

Parameters

- `ipbuf` - Char array to store the result in.
- `ip` - Internet host address in integer notation.

Return value

On success, this call returns 0 if the conversion was successful. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
ret = pico_ipv4_to_string(buf, ip);
```

3.1.2 pico_string_to_ipv4

Description

Convert the IPv4 dotted-decimal notation into binary form. The result is stored in the `int` that IP points to. Little endian or big endian is not taken into account. The address supplied in `ipstr` can have one of the following forms: a.b.c.d, a.b.c or a.b.

Function prototype

```
int pico_string_to_ipv4(const char *ipstr, uint32_t *ip);
```

Parameters

- `ipstr` - Pointer to the IP string.
- `ip` - Int pointer to store the result in.

Return value

On success, this call returns 0 if the conversion was successful. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
ret = pico_string_to_ipv4(buf, *ip);
```

3.1.3 pico_ipv4_valid_netmask

Description

Check if the provided mask if valid.

Function prototype

```
int pico_ipv4_valid_netmask(uint32_t mask);
```

Parameters

- `mask` - The netmask in integer notation.

Return value

On success, this call returns the netmask in CIDR notation is returned if the netmask is valid. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
ret = pico_ipv4_valid_netmask(netmask);
```

3.1.4 pico_ipv4_is_unicast

Description

Check if the provided address is unicast or multicast.

Function prototype

```
int pico_ipv4_is_unicast(uint32_t address);
```

Parameters

- `address` - Internet host address in integer notation.

Return value

Returns 1 if unicast, 0 if multicast.

Example

```
ret = pico_ipv4_is_unicast(address);
```

3.1.5 pico_ipv4_source_find

Description

Find the source IP belonging to the destination IP dst.

Function prototype

```
struct pico_ip4 *pico_ipv4_source_find(struct pico_ip4 *dst);
```

Parameters

- `address` - Pointer to the destination internet host address as `struct pico_ip4`.

Return value

On success, this call returns the source IP as `struct pico_ip4`. If the source can not be found, `NULL` is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EHOSTUNREACH` - host is unreachable

Example

```
src = pico_ipv4_source_find(dst);
```

3.1.6 pico_ipv4_link_add

Description

Add a new local device dev inteface, f.e. eth0, with IP address 'address' and netmask 'netmask'.

Function prototype

```
int pico_ipv4_link_add(struct pico_device *dev, struct pico_ip4 address,
struct pico_ip4 netmask);
```

Parameters

- `dev` - Local device.
- `address` - Pointer to the internet host address as `struct pico_ip4`.
- `netmask` - Netmask of the address.

Return value

On success, this call returns 0. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_ENETUNREACH` - network unreachable
- `PICO_ERR_EHOSTUNREACH` - host is unreachable

Example

```
ret = pico_ipv4_link_add(dev, address, netmask);
```

3.1.7 pico_ipv4_link_del

Description

Remove the local device dev interface, f.e. eth0, with IP address 'address'.

Function prototype

```
int pico_ipv4_link_del(struct pico_device *dev, struct pico_ip4 address);
```

Parameters

- `dev` - Local device.
- `address` - Pointer to the internet host address as `struct pico_ip4`.

Return value

On success, this call returns 0. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
ret = pico_ipv4_link_del(dev, address);
```

3.1.8 pico_ipv4_link_find

Description

Find the local device with IP address 'address'.

Function prototype

```
struct pico_device *pico_ipv4_link_find(struct pico_ip4 *address);
```

Parameters

- `address` - Pointer to the internet host address as `struct pico_ip4`.

Return value

On success, this call returns the local device. On error, `NULL` is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENXIO` - no such device or address

Example

```
dev = pico_ipv4_link_find(address);
```

3.1.9 pico_ipv4_nat_enable

Description

This function enables NAT functionality on the passed IPv4 link. Forwarded packets from an internal network will have the public IP address from the passed link and a translated port number for transmission on the external network. Usual operation requires at least one additional link for the internal network, which is used as a gateway for the internal hosts.

Function prototype

```
int pico_ipv4_nat_enable(struct pico_ipv4_link *link)
```

Parameters

- `link` - Pointer to a link `pico_ipv4_link`.

Return value

On success, this call returns 0. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
ret = pico_ipv4_nat_enable(&external_link);
```

3.1.10 pico_ipv4_nat_disable

Description

Disables the NAT functionality.

Function prototype

```
int pico_ipv4_nat_disable(void);
```

Return value

Always returns 0.

3.1.11 pico_ipv4_port_forward

Description

This function adds or deletes a rule in the IP forwarding table. Internally in the stack, a one-direction NAT entry will be made.

Function prototype

```
int pico_ipv4_port_forward(struct pico_ip4 pub_addr, uint16_t pub_port,
struct pico_ip4 priv_addr, uint16_t priv_port, uint8_t proto,
uint8_t persistant)
```

Parameters

- `pub_addr` - Public IP address, must be identical to the address of the external link.
- `pub_port` - Public port to be translated.
- `priv_addr` - Private IP address of the host on the internal network.
- `priv_port` - Private port of the host on the internal network.
- `proto` - Protocol identifier, see supported list below.
- `persistant` - Option for function call: create `PICO_IPV4_FORWARD_ADD` (= 1) or delete `PICO_IPV4_FORWARD_DEL` (= 0).

Protocol list

- `PICO_PROTO_ICMP4`
- `PICO_PROTO_TCP`
- `PICO_PROTO_UDP`

Return value

On success, this call 0 after a succesfull entry of the forward rule. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_EAGAIN` - not succesfull, try again

Example

```
ret = pico_ipv4_port_forward(ext_link_addr, ext_port, host_addr,
host_port, PICO_PROTO_UDP, 1);
```

3.1.12 pico_ipv4_route_add

Description

Add a new route to the destination IP address from the local device link, f.e. eth0.

Function prototype

```
int pico_ipv4_route_add(struct pico_ip4 address, struct pico_ip4 netmask,
struct pico_ip4 gateway, int metric, struct pico_ipv4_link *link);
```

Parameters

- `address` - Pointer to the destination internet host address as `struct pico_ip4`.
- `netmask` - Netmask of the address.
- `gateway` - Gateway of the address network.
- `metric` - Metric of the route.
- `link` - Local device interface. If a valid gateway is specified, this parameter is not mandatory, thus `NULL` can be used.

Return value

On success, this call returns 0. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_EHOSTUNREACH` - host is unreachable
- `PICO_ERR_ENETUNREACH` - network unreachable

Example

```
ret = pico_ipv4_route_add(dst, netmask, gateway, metric, link);
```

3.1.13 pico_ipv4_route_del

Description

Remove the route to the destination IP address from the local device link, f.e. etho0.

Function prototype

```
int pico_ipv4_route_del(struct pico_ip4 address, struct pico_ip4 netmask,
struct pico_ip4 gateway, int metric, struct pico_ipv4_link *link);
```

Parameters

- `address` - Pointer to the destination internet host address as struct `pico_ip4`.
- `netmask` - Netmask of the address.
- `gateway` - Gateway of the address network.
- `metric` - Metric of the route.
- `link` - Local device interface.

Return value

On success, this call returns 0 if the route is found. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
ret = pico_ipv4_route_del(dst, netmask, gateway, metric, link);
```

3.1.14 pico_ipv4_route_get_gateway

Description

This function gets the gateway address for the given destination IP address, if set.

Function prototype

```
struct pico_ip4 pico_ipv4_route_get_gateway(struct pico_ip4 *addr)
```

Parameters

- `address` - Pointer to the destination internet host address as struct `pico_ip4`.

Return value

On success the gateway address is returned. On error a `null` address is returned (`0.0.0.0`) and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EHOSTUNREACH` - host is unreachable

Example

```
gateway_addr = pico_ip4 pico_ipv4_route_get_gateway(&dest_addr)
```

3.1.15 int_pico_icmp4_ping

Description

This function sends out a number of ping echo requests and checks if the replies are received correctly. The information from the replies is passed to the callback function after a succesfull reception. If a timeout expires before a reply is received, the callback is called with the error condition.

Function prototype

```
int pico_icmp4_ping(char *dst, int count, int interval, int timeout, int size,
void (*cb)(struct pico_icmp4_stats *));
```

Parameters

- `dst` - Pointer to the destination internet host address as text string
- `count` - Number of pings going to be send
- `interval` - Time between two transmissions (in ms)
- `timeout` - Timeout period untill reply received (in ms)
- `size` - Size of data buffer in bytes
- `cb` - Callback for ICMP ping

Data structure `struct pico_icmp4_stats`

```
struct pico_icmp4_stats
{
  struct pico_ip4 dst;
  unsigned long size;
  unsigned long seq;
  unsigned long time;
  unsigned long ttl;
  int err;
};
```

With **err** values:

- `PICO_PING_ERR_REPLIED` (value 0)
- `PICO_PING_ERR_TIMEOUT` (value 1)
- `PICO_PING_ERR_UNREACH` (value 2)
- `PICO_PING_ERR_PENDING` (value 0xFFFF)

Return value

On success, this call returns 0. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENOMEM` - not enough space

Example

```
ret = pico_icmp4_ping(dst_addr, 30, 10, 100, 1000, callback);
```

3.1.16 pico_ipv4_frame_push

Description

Add an IP header to the `pico_frame f` (with destination IP `dst` and protocol `proto`) and queue the `pico_frame` to the data link layer.

Function prototype

```
int pico_ipv4_frame_push(struct pico_frame *f, struct pico_ip4 *dst, uint8_t proto);
```

Parameters

- `f` - Pointer to the frame that flows through the stack.
- `dst` - Pointer to the destination internet host address as `struct pico_ip4`.
- `proto` - IP protocol to use.

Return value

On success the new queue size is returned. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EHOSTUNREACH` - host is unreachable

Example

```
ret = pico_ipv4_frame_push(frame, dst, protocol);
```

3.1.17 pico_ipv4_rebound

Description

Rebound the frame f back to the source.

Function prototype

```
int pico_ipv4_rebound(struct pico_frame *f);
```

Parameters

- `f` - Pointer to the frame that flows through the stack

Return value

On success, this call returns the frame queue size. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EHOSTUNREACH` - host is unreachable

Example

```
ret = pico_ipv4_rebound(frame);
```

3.1.18 dbg_route

Description

Print the complete routing table.

Function prototype

```
void dbg_route(void);
```

3.2 Socket calls

With the socket calls, the user can open, close, bind, ... sockets and do read or write operations. The provided transport protocols are UDP and TCP.

3.2.1 pico_socket_open

Description

This function will be called to open a socket from the application level. The created socket will be unbound.

Function prototype

```
struct pico_socket *pico_socket_open(uint16_t net, uint16_t proto,
void (*wakeup)(uint16_t ev, struct pico_socket *s));
```

Parameters

- `net` - Network protocol, `PICO_PROTO_IPV4` = 0, `PICO_PROTO_IPV6` = 41
- `proto` - Transport protocol, `PICO_PROTO_TCP` = 6, `PICO_PROTO_UDP` = 17
- `wakeup` - Callback function that accepts 2 parameters:
  - `ev` - Events that apply to that specific socket, see further
  - `s` - Pointer to a socket of type struct `pico_socket`

Possible events for sockets

- `PICO_SOCK_EV_RD` - trigerred when data arrived on the socket
- `PICO_SOCK_EV_WR` - trigerred when ready to write to the socket (TCP only)
- `PICO_SOCK_EV_CONN` - trigerred when connection is established (TCP only)

- `PICO_SOCK_EV_CLOSE` - trigerred when FIN packet received (TCP only)
- `PICO_SOCK_EV_FIN` - trigerred when the socket is closed (TCP only)
- `PICO_SOCK_EV_ERR` - trigerred when an error occurs

Return value

On success, this call returns a pointer to the declared socket (`struct pico_socket *`). On error the socket is not created, `NULL` is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EPROTONOSUPPORT` - protocol not supported
- `PICO_ERR_ENETUNREACH` - network unreachable

Example

```
sk_tcp = pico_socket_open(PICO_PROTO_IPV4, PICO_PROTO_TCP, &wakeup);
```

3.2.2 pico_socket_read

Description

This function will be called to read a string from a socket from the application level. The function checks whether or not the socket is bound.

Function prototype

```
int pico_socket_read(struct pico_socket *s, void *buf, int len);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `buf` - Void pointer to the start of a string buffer where the string will be stored
- `len` - Length of the string that was read from the socket (in bytes)

Return value

On success, this call returns an integer representing the number of bytes read. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EIO` - input/output error
- `PICO_ERR_ESHUTDOWN` - cannot read after transport endpoint shutdown

Example

```
bytesRead = pico_socket_read(sk_tcp, buffer, bufferLength);
```

3.2.3 pico_socket_write

Description

This function will be called to write a string to a socket from the application level. This function also checks if the socket is bound, connected and that it isn't shutdown locally. This is the preferred function to use when writing strings from application level.

Function prototype

```
int pico_socket_write(struct pico_socket *s, void *buf, int len);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `buf` - Void pointer to the start of a string buffer where the string is stored
- `len` - Length of the string that is stored in the buffer (in bytes)

Return value

On success, this call returns an integer representing the number of bytes written to the socket. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EIO` - input/output error
- `PICO_ERR_ENOTCONN` - the socket is not connected
- `PICO_ERR_ESHUTDOWN` - cannot send after transport endpoint shutdown
- `PICO_ERR_EADDRNOTAVAIL` - address not available
- `PICO_ERR_EHOSTUNREACH` - host is unreachable
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_EAGAIN` - resource temporarily unavailable

Example

```
bytesWritten = pico_socket_write(sk_tcp, buffer, bufLength);
```

3.2.4 pico_socket_sendto

Description

This function is be called by the `pico_socket_write` and `pico_socket_send` functions. This function sends a string from the local address to the remote address, without checking if the remote is connected or not.

Function prototype

```
int pico_socket_sendto(struct pico_socket *s, void *buf, int len,
void *dst, uint16_t remote_port);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `buf` - Void pointer to the start of a string buffer where the string is stored
- `len` - Length of the string that is stored in the buffer (in bytes)
- `dst` - Pointer to the origin of the IPv4/IPv6 frame header
- `remote_port` - Portnumber of the receiving socket

Return value

On success, this call returns an integer representing the number of bytes written to the socket. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EADDRNOTAVAIL` - address not available
- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EHOSTUNREACH` - host is unreachable
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_EAGAIN` - resource temporarily unavailable

Example

```
bytesWritten = pico_socket_sendto(sk_tcp, buf, len, &sk_tcp->remote_addr,
sk_tcp->remote_port);
```

3.2.5 pico_socket_recvfrom

Description

This function is called to receive a string of data from the specified socket. This function also checks if the socket is bound but not if it is connected or shutdown locally.

Function prototype

```
int pico_socket_recvfrom(struct pico_socket *s, void *buf, int len,
void *orig, uint16_t *remote_port);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `buf` - Void pointer to the start of a string buffer where the string will be stored
- `len` - Length of the string that will be stored in the buffer (in bytes)
- `orig` - Pointer to the origin of the IPv4/IPv6 frame header
- `remote_port` - Portnumber of the sender socket (pointer)

Return value

On success, this call returns an integer representing the number of bytes read from the socket. Also `remote_port` will contain the portnumber of the sending socket. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ESHUTDOWN` - cannot read after transport endpoint shutdown
- `PICO_ERR_EADDRNOTAVAIL` - address not available

Example

```
bytesRcvd = pico_socket_recvfrom(sk_tcp, buf, bufLen, &peer, &port);
```

3.2.6 pico_socket_send

Description

This function is called to send a string of data to the specified socket. This function also checks if the socket is connected and then calls the `pico_socket_sendto` function.

Function prototype

```
int pico_socket_send(struct pico_socket *s, void *buf, int len);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `buf` - Void pointer to the start of a string buffer where the string is stored
- `len` - Length of the string that is stored in the buffer (in bytes)

Return value

On success, this call returns an integer representing the number of bytes written to the socket. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENOTCONN` - the socket is not connected
- `PICO_ERR_EADDRNOTAVAIL` - address not available
- `PICO_ERR_EHOSTUNREACH` - host is unreachable
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_EAGAIN` - resource temporarily unavailable

Example

```
bytesRcvd = pico_socket_send(sk_tcp, buf, bufLen);
```

3.2.7 pico_socket_recv

Description

This function directly calls the `pico_socket_recvfrom` function.

Function prototype

```
int pico_socket_recv(struct pico_socket *s, void *buf, int len);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `buf` - Void pointer to the start of a string buffer where the string will be stored
- `len` - Length of the string in the socket buffer (in bytes)

Return value

On success, this call returns an integer representing the number of bytes read from the socket. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ESHUTDOWN` - cannot read after transport endpoint shutdown
- `PICO_ERR_EADDRNOTAVAIL` - address not available

Example

```
bytesRcvd = pico_socket_recv(sk_tcp, buf, bufLen);
```

3.2.8 pico_socket_bind

Description

This function binds a local IP-address and port to the specified socket.

Function prototype

```
int pico_socket_bind(struct pico_socket *s, void *local_addr, uint16_t *port);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `local_addr` - Void pointer to the local IP-address
- `port` - Local portnumber to bind with the socket

Return value

On success, this call returns 0 after a succesfull bind. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_ENXIO` - no such device or address

Example

```
errMsg = pico_socket_bind(sk_tcp, &sockaddr4->addr, &sockaddr4->port);
```

3.2.9 pico_socket_connect

Description

This function connects a local socket to a remote socket of a server that is listening.

Function prototype

```
int pico_socket_connect(struct pico_socket *s, void *srv_addr,
uint16_t remote_port);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `srv_addr` - Void pointer to the remote IP-address to connect to
- `remote_port` - Remote port number on which the socket will be connected to

Return value

On success, this call returns 0 after a succesfull connect. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EPROTONOSUPPORT` - protocol not supported
- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EHOSTUNREACH` - host is unreachable

Example

```
errMsg = pico_socket_connect(sk_tcp, &sockaddr4->addr, sockaddr4->port);
```

3.2.10 pico_socket_listen

Description

A server can use this function when a socket is opened and bound to start listening to it.

Function prototype

```
int pico_socket_listen(struct pico_socket *s, int backlog);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `backlog` - Maximum connection requests

Return value

On success, this call returns 0 after a succesfull listen start. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EISCONN` - socket is connected

Example

```
errMsg = pico_socket_listen(sk_tcp, 3);
```

3.2.11 pico_socket_accept

Description

When a server is listening on a socket and the client is trying to connect. The server on his side will wakeup and acknowledge the connection by calling the this function.

Function prototype

```
struct pico_socket *pico_socket_accept(struct pico_socket *s, void *orig,
uint16_t *local_port);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `orig` - Pointer to the origin of the IPv4/IPv6 frame header
- `local_port` - Portnumber of the local socket (pointer)

Return value

On success, this call returns the pointer to a `struct pico_socket` that represents the client thas was just connected. Also `orig` will contain the requesting IP-address and `remote_port` will contain the portnumber of the requesting socket. On error, `NULL` is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EAGAIN` - resource temporarily unavailable

Example

```
client = pico_socket_accept(sk_tcp, &peer, &port);
```

3.2.12 pico_socket_shutdown

Description

Used by the `pico_socket_close` function to shutdown read and write mode for the specified socket. With this function one can close a socket for reading and/or writing.

Function prototype

```
int pico_socket_shutdown(struct pico_socket *s, int mode);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `mode` - `PICO_SHUT_RDWR`, `PICO_SHUT_WR`, `PICO_SHUT_RD`

Return value

On success, this call returns 0 after a succesfull socket shutdown. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
errMsg = pico_socket_shutdown(s, PICO_SHUT_RDWR);
```

3.2.13 pico_socket_close

Description

Function used on application level to close a socket. Always closes read and write connection.

Function prototype

```
int pico_socket_close(struct pico_socket *s);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`

Return value

On success, this call returns 0 after a succesfull socket shutdown. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
errMsg = pico_socket_close(sk_tcp);
```

3.2.14 pico_socket_setoption

Description

Function used to set socket options.

Function prototype

```
int pico_socket_setoption(struct pico_socket *s, int option, void *value);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `option` - Option to be set (see further for all options)
- `value` - Value of option (void pointer)

Available socket options

- `PICO_TCP_NODELAY` - Disables the Nagle algorithm (`value` not used)
- `PICO_IP_MULTICAST_IF` - (Not supported) Set link multicast datagrams are sent from, default is first added link
- `PICO_IP_MULTICAST_TTL` - Set TTL (0-255) of multicast datagrams, default is 1
- `PICO_IP_MULTICAST_LOOP` - Specifies if a copy of an outgoing multicast datagram is looped back as long as it is a member of the multicast group, default is enabled
- `PICO_IP_ADD_MEMBERSHIP` - Join the multicast group specified
- `PICO_IP_DROP_MEMBERSHIP` - Leave the multicast group specified

Return value

On success, this call returns 0 after a succesfull setting of socket option. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
ret = pico_socket_setoption(sk_tcp, PICO_TCP_NODELAY, NULL);

uint8_t ttl = 2;
ret = pico_socket_setoption(sk_udp, PICO_IP_MULTICAST_TTL, &ttl);

uint8_t loop = 0;
ret = pico_socket_setoption(sk_udp, PICO_IP_MULTICAST_LOOP, &loop);

struct pico_ip4 inaddr_dst, inaddr_link;
```

```
struct pico_ip_mreq mreq = {{0},{0}};
pico_string_to_ipv4("224.7.7.7", &inaddr_dst.addr);
pico_string_to_ipv4("192.168.0.2", &inaddr_link.addr);
mreq.mcast_group_addr = inaddr_dst;
mreq.mcast_link_addr = inaddr_link;
ret = pico_socket_setoption(sk_udp, PICO_IP_ADD_MEMBERSHIP, &mreq);
ret = pico_socket_setoption(sk_udp, PICO_IP_DROP_MEMBERSHIP, &mreq)
```

3.2.15 pico_socket_getoption

Description

Function used to get socket options.

Function prototype

```
int pico_socket_getoption(struct pico_socket *s, int option, void *value);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `option` - Option to be set (see further for all options)
- `value` - Value of option (void pointer)

Available socket options

- `PICO_TCP_NODELAY` - Nagle algorithm, `value` casted to `(int *)` (0 = disabled, 1 = enabled)
- `PICO_IP_MULTICAST_IF` - (Not supported) Link multicast datagrams are sent from
- `PICO_IP_MULTICAST_TTL` - TTL (0-255) of multicast datagrams
- `PICO_IP_MULTICAST_LOOP` - Loop back a copy of an outgoing multicast datagram, as long as it is a member of the multicast group, or not.

Return value

On success, this call returns 0 after a succesfull getting of socket option. The value of the option is written to `value`. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
ret = pico_socket_getoption(sk_tcp, PICO_TCP_NODELAY, &stat);

uint8_t ttl = 0;
ret = pico_socket_getoption(sk_udp, PICO_IP_MULTICAST_TTL, &ttl);

uint8_t loop = 0;
ret = pico_socket_getoption(sk_udp, PICO_IP_MULTICAST_LOOP, &loop);
```

3.3 DHCP client

A DHCP client for obtaining a dynamic IP address. When initiating a negotiation the user is passed an identifier, which must then be passed to all future calls to `pico_dhcp` functions. (Currently DHCP can only be run on one interface. Future versions may support DHCP on multiple interfaces, and the functions described here are already prepared for that.)

3.3.1 pico_dhcp_initiate_negotiation

Description

Initiate a DHCP negotiation. The user passes a callback-function, which will be called when DHCP has succeeded or failed.

Function prototype

```
void * pico_dhcp_initiate_negotiation(struct pico_device* device,
void (*callback)(void* cli, int code));
```

Parameters

- `device` - the device on which a negotiation should be started
- `callback` - the function which will be called in case of success or failure. Note that this function can be called multiple times. An example would be if initially DHCP succeeded, but then the DHCP server was removed from the network long enough for the lease to expire, and later added again to the network. The callback would be called 3 times in this example: first with code `PICO_DHCP_SUCCESS`, then with `PICO_DHCP_RESET`, and finally again with `PICO_DHCP_SUCCESS`. Also note that this callback may already be called before `pico_dhcp_initiate_negotiation` has returned, e.g. in case of failure to open a socket. It accepts two parameters :
  - `cli` - the identifier of the negotiation
  - `code` - the reason the callback occurred, see further

Possible DHCP codes

- `PICO_DHCP_SUCCESS` - DHCP succeeded, the user can start using the assigned address, which can be obtained by calling `pico_dhcp_get_address`.
- `PICO_DHCP_ERROR` - an error occurred. DHCP is unable to recover from this error. `pico_err` is set appropriately.
- `PICO_DHCP_RESET` - DHCP was unable to renew its lease, and the lease expired. The user must immediately stop using the previously assigned IP, and wait for DHCP to obtain a new lease. DHCP will automatically start negotiations again.

Return value

A `void*` identifying the negotiation. This must be passed to all calls related to DHCP. This is to create the possibility of initiating DHCP negotiations on multiple devices (currently not supported).

Errors

All errors are reported through the callback-function described above.

- `PICO_ERR_EADDRNOTAVAIL` - address not available

- PICO_ERR_EINVAL - invalid argument
- PICO_ERR_EHOSTUNREACH - host is unreachable
- PICO_ERR_ENOMEM - not enough space
- PICO_ERR_EAGAIN - resource temporarily unavailable
- PICO_ERR_EPROTONOSUPPORT - protocol not supported
- PICO_ERR_ENETUNREACH - network unreachable
- PICO_ERR_EINVAL - invalid argument
- PICO_ERR_ENXIO - no such device or address
- PICO_ERR_EOPNOTSUPP - operation not supported on socket

Example

```
void* identifier = pico_dhcp_initiate_negotiation(dev, &callback_dhcpclient);
```

3.3.2 pico_dhcp_get_address

Description

Get the address that was assigned through DHCP. This function should only be called after a callback occurred with code PICO_DHCP_SUCCESS.

Function prototype

```
struct pico_ip4 pico_dhcp_get_address(void* cli);
```

Parameters

- cli - the negotiation identifier that was returned from pico_dhcp_initiate_negotiations.

Return value

struct pico_ip4 - the address that was assigned

Example

```
struct pico_ip4 address = pico_dhcp_get_address(identifier);
```

3.3.3 pico_dhcp_get_gateway

Description

Get the address of the gateway that was assigned through DHCP. This function should only be called after a callback occurred with code PICO_DHCP_SUCCESS.

Function prototype

```
struct pico_ip4 pico_dhcp_get_gateway(void* cli);
```

Parameters

- cli : the negotiation identifier that was returned from pico_dhcp_initiate_negotiations.

Return value

- struct pico_ip4 - the address of the gateway that should be used.

Example

```
struct pico_ip4 gateway = pico_dhcp_get_gateway(identifier);
```

3.4 DHCP server

3.4.1 pico_dhcp_server_initiate

Description

This function starts a simple DHCP server.

Function prototype

```
int pico_dhcp_server_initiate(struct pico_dhcpd_settings* settings);
```

Parameters

- `settings` - a pointer to a struct `pico_dhcpd_settings`, in which the following members matter to the user :
  - `struct pico_device *dev` - a pointer to the device on which the dhcp server must operate
  - `struct pico_ip4 my_ip` - the IP assigned to the server
  - `struct pico_ip4 netmask` - the netmask the server must advertise
  - `uint32_t pool_start` - the first IP address that may be assigned
  - `uint32_t pool_end` - the last IP address that may be assigned
  - `uint32_t lease_time` - the advertised lease time in seconds

Return value

On successful startup of the dhcp server, 0 is returned. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- PICO_ERR_EPROTONOSUPPORT - protocol not supported
- PICO_ERR_ENETUNREACH - network unreachable
- PICO_ERR_EINVAL - invalid argument
- PICO_ERR_ENXIO - no such device or address

Example

```
struct pico_dhcpd_settings s = {0};
s.dev = ethernet;
s.my_ip.addr = long_be(0x0a280003);
s.netmask.addr = long_be(0xffffff00);
s.pool_start = (s.my_ip.addr & long_be(0xffffff00)) | long_be(0x00000064);
s.pool_end = (s.my_ip.addr & long_be(0xffffff00)) | long_be(0x000000ff);
pico_dhcp_server_initiate(&s);
```

3.5 DNS client

3.5.1 pico_dns_client_nameserver

Description

Function to add or remove nameservers.

Function prototype

```
int pico_dns_client_nameserver(struct pico_ip4 *ns, uint8_t flag);
```

Parameters

- `ns` - Pointer to the address of the name server.
- `flag` - Flag to indicate addition or removal (see further).

Flags

- `PICO_DNS_NS_ADD` - to add a nameserver
- `PICO_DNS_NS_DEL` - to remove a nameserver

Return value

On success, this call returns 0 if the nameserver operation has succeeded. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_EAGAIN` - resource temporarily unavailable

Example

```
ret = pico_dns_client_nameserver(&addr_ns, PICO_DNS_NS_ADD);
ret = pico_dns_client_nameserver(&addr_ns, PICO_DNS_NS_DEL);
```

3.5.2 pico_dns_client_getaddr

Description

Function to translate an url text string to an internet host address IP.

Function prototype

```
int pico_dns_client_getaddr(const char *url, void (*callback)(char *ip));
```

Parameters

- `url` - Pointer to text string containing url text string (e.g. www.google.com)
- `calback` - Callback function, receiving the internet host address IP. Note: the returned string has to be freed by the user.

Return value

On success, this call returns 0 if the request is sent. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_EAGAIN` - resource temporarily unavailable

Example

```
int ret = pico_dns_client_getaddr("www.google.com", cb_getaddr);
```

3.5.3 pico_dns_client_getname

Description

Function to translate an internet host address IP to an url text string.

Function prototype

```
int pico_dns_client_getname(const char *ip, void (*callback)(char *url));
```

Parameters

- `ip` - Pointer to text string containing an internet host address IP (e.g. 8.8.4.4)
- `callback` - Callback function, receiving the url text string. Note: the returned string has to be freed by the user.

Return value

On success, this call returns 0 if the request is sent. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_ENOMEM` - not enough space
- `PICO_ERR_EAGAIN` - resource temporarily unavailable

Example

```
int ret = pico_dns_client_getname("8.8.4.4", cb_getname);
```

3.6 IGMP

This module allows the user to join and leave IGMP multicast groups. Currently only IGMP2 is supported.

3.6.1 pico_igmp2_join_group

Description

Join an IGMP2 multicast group.

Function prototype

```
int pico_igmp2_join_group(struct pico_ip4 *group_address,
struct pico_ipv4_link *link);
```

Parameters

- `group_address` - the address of the multicast group you want to join.
- `link` - the link on which that multicast group should be joined.

Errors

In case of failure, -1 is returned, and the value of pico_err is set as follows:

- `PICO_ERR_EINVAL` - Invalid argument provided
- `PICO_ERR_EFAULT` - Internal error
- `PICO_ERR_EEXIST` - Attempted to join a group that was already joined before

3.6.2 pico_igmp2_leave_group

Description

leave an IGMP2 multicast group.

Function prototype

```
int pico_igmp2_leave_group(struct pico_ip4 *group_address, struct pico_ipv4_link *link);
```

Parameters

- `group_address` - the address of the multicast group you want to leave.
- `link` - the link on which that multicast group should be left.

Return value

In case of success, 0. In case of failure, -1 is returned and pico_err is set accordingly.

Errors

In case of success, zero is returned. In case of failure, -1 is returned, and the value of pico_err is set as follows:

- `PICO_ERR_EINVAL` - Invalid argument provided
- `PICO_ERR_EFAULT` - Internal error
- `PICO_ERR_ENOENT` - Attempted to leave a group which has never been joined

3.7 IP Filter

This module allows the user to add and remove filters. The user can filter packets based on interface, protocol, outgoing address, outgoing netmask, incomming address, incomming netmask, outgoing port, incomming port, priority and type of service. There are four types of filters: ACCEPT, PRIORITY, REJECT, DROP. When creating a PRIORITY filter, it is necessary to give a priority value in a range between '-10' and '10', '0' as default priority.

3.7.1 pico_ipv4_filter_add

Description

Function to add a filter.

Function prototype

```
int pico_ipv4_filter_add(struct pico_device *dev, uint8_t proto,
  struct pico_ip4 out_addr, struct pico_ip4 out_addr_netmask,
  struct pico_ip4 in_addr, struct pico_ip4 in_addr_netmask, uint16_t out_port,
  uint16_t in_port, int8_t priority, uint8_t tos, enum filter_action action);
```

Parameters

- `dev` - interface to be filtered
- `proto` - protocol to be filtered
- `out_addr` - outgoing address to be filtered
- `out_addr_netmask` - outgoing address-netmask to be filtered
- `in_addr` - incomming address to be filtered
- `in_addr_netmask` - incomming address-netmask to be filtered
- `out_port` - outgoing port to be filtered
- `in_port` - incomming port to be filtered
- `priority` - priority to be filtered
- `tos` - type of service to be filtered
- `action` - type of action for the filter: ACCEPT, PRIORITY, REJECT and DROP. ACCEPT, filters all packets selected by the filter. PRIORITY is not yet implemented. REJECT drops all packets and send an ICMP message 'Packet Filtered' (Communication Administratively Prohibited). DROP will discard the packet silently.

Return value

On success, this call returns the filter_id from the generated filter. This id must be used when deleting the filter. On error, -1 is returned and `pico_err` is set appropriately.

Example

```
/* block all incoming traffic on port 5555 */
filter_id = pico_ipv4_filter_add(NULL, 6, NULL, NULL, NULL, NULL, 0, 5555,
0, 0, FILTER_REJECT);
```

Errors

- `PICO_ERR_EINVAL` - invalid argument

3.7.2 pico_ipv4_filter_del

Description

Function to delete a filter.

Function prototype

```
int pico_ipv4_filter_del(int filter_id)
```

Parameters

- `filter_id` - the id of the filter you want to delete.

Return value

On success, this call returns 0. On error, -1 is returned and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument
- `PICO_ERR_EPERM` - operation not permitted

Example

```
ret = pico_ipv4_filter_del(filter_id);
```

4. Examples

The following sections will give code examples of PicoTCP. It is assumed that all examples include the appropriate header files and a **main** routine that calls the `app_x` functions to initialize the example.

The most common header files are:

```
#include "pico_stack.h"
#include "pico_config.h"
#include "pico_dev_vde.h"
#include "pico_ipv4.h"
#include "pico_socket.h"
#include "pico_dev_tun.h"
#include "pico_nat.h"
#include "pico_icmp4.h"
#include "pico_dns_client.h"
#include "pico_dev_loop.h"
#include "pico_dhcp_client.h"
#include "pico_dhcp_server.h"
#include "pico_ipfilter.h"
```

4.1 Ping example

```
#define NUM_PING 10

/* callback function for receiving ping reply */
void cb_ping(struct pico_icmp4_stats *s)
{
  char host[30];
  int time_sec = 0;
  int time_msec = 0;

  /* convert ip address from icmp4_stats structure to string */
  pico_ipv4_to_string(host, s->dst.addr);

  /* get time information from icmp4_stats structure */
  time_sec = s->time / 1000;
  time_msec = s->time % 1000;

  if (s->err == PICO_PING_ERR_REPLIED) {
   /* print info if no error reported in icmp4_stats structure */
    dbg("%lu bytes from %s: icmp_req=%lu ttl=%lu time=%lu ms\n", \
     s->size, host, s->seq, s->ttl, s->time);
    if (s->seq >= NUM_PING)
      exit(0);
```

```
  } else {
   /* else, print error info */
    dbg("PING %lu to %s: Error %d\n", s->seq, host, s->err);
    exit(1);
  }
}

/* initialize the ping command */
void app_ping(char *dest)
{
  pico_icmp4_ping(dest, NUM_PING, 1000, 5000, 48, cb_ping);
}
```

4.2 UDP echo socket example

```
struct pico_ip4 inaddr_any = { };

/* callback for UDP echo socket events */
void cb_udpecho(uint16_t ev, struct pico_socket *s)
{
  char recvbuf[1400];
  int read = 0;
  uint32_t peer;
  uint16_t port;

  /* process read event, data available */
  if (ev == PICO_SOCK_EV_RD) {
   /* while data available in socket buffer, echo data to peer */
    do {
      read = pico_socket_recvfrom(s, recvbuf, 1400, &peer, &port);
      if (read > 0)
        pico_socket_sendto(s, recvbuf, r, &peer, port);
    } while(read > 0);
  }

  /* process error event, socket error occured */
  if (ev == PICO_SOCK_EV_ERR) {
    printf("Socket Error received. Bailing out.\n");
    exit(1);
  }

  printf("Received data from %08X:%u\n", peer, port);
}

/* initialize the UDP echo socket */
void app_udpecho(uint16_t source_port)
{
  struct pico_socket *s;
```

```
  uint16_t port_be = 0;

  /* set the source port for the socket */
  if (source_port == 0)
    port_be = short_be(5555);
  else
    port_be = short_be(source_port);

  /* open a UDP socket with the appropriate callback */
  s = pico_socket_open(PICO_PROTO_IPV4, PICO_PROTO_UDP, &cb_udpecho);
  if (!s)
    exit(1);

  /* bind the socket to port_be */
  if (pico_socket_bind(s, &inaddr_any, &port_be) != 0)
    exit(1);
}
```

4.3 TCP echo socket example

```
#define BSIZE 1460

/* callback for TCP echo socket events */
void cb_tcpecho(uint16_t ev, struct pico_socket *s)
{
  char recvbuf[BSIZE];
  int read = 0, written = 0;
  int pos = 0, len = 0;
  struct pico_socket *sock_a;
  struct pico_ip4 orig;
  uint16_t port;
  char peer[30];

  /* process read event, data available */
  if (ev & PICO_SOCK_EV_RD) {
    do {
      read = pico_socket_read(s, recvbuf + len, BSIZE - len);
      if (read > 0)
        len += r;
    } while(read > 0);
  }

  /* process connect event, syn received */
  if (ev & PICO_SOCK_EV_CONN) {
    /* accept new connection request */
    sock_a = pico_socket_accept(s, &orig, &port);

    /* convert peer IP to string */
```

```
    pico_ipv4_to_string(peer, orig.addr);

    /* print info */
    printf("Connection established with %s:%d.\n", peer, short_be(port));
  }

  /* process fin event, receiving socket closed */
  if (ev & PICO_SOCK_EV_FIN) {
    printf("Socket closed. Exit normally. \n");
  }

  /* process error event, socket error occured */
  if (ev & PICO_SOCK_EV_ERR) {
    printf("Socket Error received: %s. Bailing out.\n", strerror(pico_err));
    exit(1);
  }

  /* process close event, receiving socket received close from peer */
  if (ev & PICO_SOCK_EV_CLOSE) {
    printf("Socket received close from peer.\n");
    /* shutdown write side of socket */
    pico_socket_shutdown(s, PICO_SHUT_WR);
  }

  /* if data read, echo back */
  if (len > pos) {
    do {
      /* echo data back to peer */
      written = pico_socket_write(s, recvbuf + pos, len - pos);
      if (written > 0) {
        pos += written;
        if (pos >= len) {
          pos = 0;
          len = 0;
          written = 0;
        }
      } else {
        printf("SOCKET> ECHO write failed, dropped %d bytes\n",(len-pos));
      }
    } while(written > 0);
  }
}

/* initialize the TCP echo socket */
void app_tcpecho(uint16_t source_port)
{
  struct pico_socket *s;
  uint16_t port_be = 0;
  int backlog = 40; /* max number of accepting connections */
```

```
  int ret;

  /* set the source port for the socket */
  if (source_port == 0)
    port_be = short_be(5555);
  else
    port_be = short_be(source_port);

  /* open a TCP socket with the appropriate callback */
  s = pico_socket_open(PICO_PROTO_IPV4, PICO_PROTO_TCP, &cb_tcpecho);
  if (!s)
    exit(1);

  /* bind the socket to port_be */
  ret = pico_socket_bind(s, &inaddr_any, &port_be);
  if (ret != 0)
    exit(1);

  /* start listening on socket */
  ret = pico_socket_listen(s, backlog);
  if (ret != 0)
    exit(1);
}
```

4.4 NAT setup example

```
/* initialize NAT functionality and add port forward rule */
void app_nat(char *dest)
{
  char *dest = NULL;
  struct pico_ip4 ipdst, pub_addr, priv_addr;
  struct pico_ipv4_link *link;

  /* convert IP address of link where to enable NAT */
  pico_string_to_ipv4(dest, &ipdst.addr);

  /* get link pointer */
  link = pico_ipv4_link_get(&ipdst);
  if (!link) {
    printf("destination not found\n");
    exit(1);
  }

  /* enable NAT on link */
  pico_ipv4_nat_enable(link);

  /* add port forward rule */
  pico_string_to_ipv4("10.50.0.10", &pub_addr.addr);
```

```
  pico_string_to_ipv4("10.40.0.08", &priv_addr.addr);
  pico_ipv4_port_forward(pub_addr, short_be(5555), priv_addr, short_be(6667),
  PICO_PROTO_UDP, PICO_IPV4_FORWARD_ADD);

  printf("nat started\n");
}
```

4.5 DNS example

```
/* callback function for receiving URL translation */
void cb_getaddr(char *ip)
{
  /* NULL indicates an error condition */
  if (!ip) {
    printf("DNS error occured: %s\n", strerror(pico_err));
    return;
  }
  printf("DNS translation to ip %s\n", ip);

  /* important: free the received pointer! */
  pico_free(ip);
}

/* callback function for receiving IP translation */
void cb_getname(char *url)
{
  /* NULL indicates an error condition */
  if (!url) {
    printf("DNS error occured: %s\n", strerror(pico_err));
    return;
  }
  printf("DNS translation to url %s\n", url);

  /* important: free the received pointer! */
  pico_free(url);
}

/* initialize the dns */
void app_dns(char *url, char *ip)
{
  struct pico_ip4 nameserver;

  /* optional: add custom dns nameserver */
  pico_string_to_ipv4("8.8.4.4", &nameserver.addr);
  pico_dns_client_nameserver(&nameserver, PICO_DNS_NS_ADD);

  /* request translation of URL f.e. www.google.com */
  pico_dns_client_getaddr(url, &cb_getaddr);
```

```
  /* request translation of IP f.e. 8.8.8.8 */
  pico_dns_client_getname(ip, &cb_getname);
}
```

A. Supported RFC's

RFC	Description
RFC 793	This RFC describes the TCP standard. The following requirements are facilitated: (1) Basic Data Transfer, (2) Reliability, (3) Flow Control, (4) Multiplexing, (5) Connection Management.
RFC 813	This RFC describes the implementation of (1) the acknowledgement mechanism and (2) window mechanism (flow control).
RFC 817	This RFC will discuss some of the commonly encountered reasons why protocol implementations seem to run slowly. Two aspects to achieve good protocol performance are described: (1) how the implementation of the protocol is integrated in an OS (scheduling, resources, interrupts, ...) and (2) how the protocol package itself is organized internally (packet size, unneeded packets, ...)
RFC 872	This RFC discusses the position that the usage of TCP and IP on LAN's is inappropriate. The conclusion is that the sometimes-expressed fear that using TCP on a local net is a bad idea is unfounded.
RFC 879	This RFC discusses the TCP Maximum Segment Size Option. The TCP maximum segment size (MSS) can be calculated depending on the network MTU, or it can be communicated by a TCP option.
RFC 896	This RFC discusses some aspects of congestion control in IP/TCP Internetworks. The Nagle algorithm suggests that the sending of new data should inhibited when there remain unacknowledged packets. When packets are acknowledged, new packets in the buffer can be transmitted (until the window size). This scheme reduces the amount of small packets transmitted.
RFC 964	This note points out three errors with the specification of the Military Standard Transmission Control Protocol (MIL-STD-1778). The following problems are discussed: (1) data accompanying a SYN can not be accepted because of errors in the acceptance policy, (2) no retransmission timer is set for a SYN packet, and therefore the SYN will not be retransmitted if it is lost, (3) when the connection has been established, neither entity takes the proper steps to accept incoming data.
RFC 1071	This RFC gives an overview of methods for efficiently computing the Internet checksum that is used by the standard Internet protocols (1) IP, (2) UDP, and TCP.
RFC 1106	This RFC discusses two extensions to the TCP protocol to provide a more efficient operation over a network with a high bandwidth*delay product: (1) NAK Option, (2) Big Windows.
RFC 1122	Requirements for Internet Hosts – Communication Layers
RFC 1180	This RFC is a tutorial on the TCP/IP protocol suite, focusing particularly on the steps in forwarding an IP datagram from source host to destination host through a router.

RFC 1263	This RFC comments on recent proposals to extend TCP (see RFC 1072 and RFC 1185). The costs and benefits of three approaches to making these changes are compared: (1) the creation of new protocols, (2) backward compatible protocol extensions and (3) protocol evolution.
RFC 1323	This memo presents a set of TCP extensions to improve performance over large bandwidth*delay product paths and to provide reliable operation over very high-speed paths. It defines new TCP options for scaled windows and timestamps, which are designed to provide compatible interworking with TCP's that do not implement the extensions. The timestamps are used for two distinct mechanisms: RTTM (Round Trip Time Measurement) and PAWS (Protect Against Wrapped Sequences). Selective acknowledgments are not included in this memo. This memo combines and supersedes RFC 1072 and RFC 1185, adding additional clarification and more detailed specification.
RFC 1337	This note describes some theoretically-possible failure modes for TCP connections and discusses possible remedies. Especially the TIME-WAIT Assassination" (TWA) problem and solution are discussed.
RFC 2018	TCP Selective Acknowledgment Options
RFC 2131	Dynamic Host Configuration Protocol
RFC 2132	DHCP Options and BOOTP Vendor Extensions
RFC 2236	IGMPv2: Host functionality implemented, Router functionality NOT implemented
RFC 2581	This RFC defines TCP's four intertwined congestion control algorithms: (1) slow start, (2) congestion avoidance, (3) fast retransmit and (4) fast recovery.
RFC 2663	This RFC describes NAT. The implemented NAT method is NAPT.