picoTCP User Documentation

December 16, 2015

December 16, 2015

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.

- **MDNS client** picoTCP has a mDNS responder on which records can be registered and resolved without the need of a centralised DNS-server. Supports possible caching of records and defending of hostnames.
- **DNS-SD client** The stack can register services on the network via Multicast DNS, which allows Zero Configuration Networking capabilities.
- **SNTP client** The stack supports synchronizing time over the network using sntp with a precision of at least 10msec. Similar to unix a gettimeofday function is implemented.
- **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.
PROFILE	0,1	0	...
PERF	0,1	0	...
STRIP	0,1	0	...
RTOS	0,1	0	...
GENERIC	0,1	0	...
PTHREAD	0,1	0	...
ADDRESS_SANITIZER	0,1	1	...
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.
ETH	0,1	1	...
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.
IPV4FRAG	0,1	1	Enables the support for fragmentation and reassembly of IPV4 packets.
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).

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.
SLAACV4	0,1	1	...
IPV6	0,1	1	Enables the support for IPV6 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.
IPV6FRAG	0,1	0	Enables the support for fragmentation and reassembly of IPV6 packets.
CRC	0,1	1	If enabled, CRC values are validated at IP and transport layer.
IPFILTER	0,1	1	If enabled, provides basic filtering.
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.
DNS_CLIENT	0,1	1	This feature is required to resolve host names into IP addresses and vice-versa.
MDNS	0,1	1	If enabled, registering and resolving DNS records on the network via Multicast DNS is possible.
DNS_SD	0,1	1	If enabled, it is possible to register services on the network via Multicast DNS.
SNTP_CLIENT	0,1	1	Enables snychronising the local time to a given ntp server.
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.
HTTP_CLIENT	0,1	1	Activates a basic HTTP client.
HTTP_SERVER	0,1	1	Activates a basic HTTP server.
OSLR	0,1	0	...
TFTP	0,1	1	...
AODV	0,1	1	...
MEMORY_MANAGER	0,1	0	...
MEMORY_MANAGER_PROFILING	0,1	0	...
TUN	0,1	0	...
TAP	0,1	0	...
PCAP	0,1	0	...
PPP	0,1	1	...
IPC	0,1	0	...
CYASSL	0,1	0	...
WOLFSSL	0,1	0	...
POLARSSL	0,1	0	...

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"
stm32f1xx	ARM Cortex M3	muRata SN820X
stellaris	ARM Cortex LM3S-6965	Texas Instrument Evaluation Kit "Codesourcery LM3S6965 ETH"
lpc18xx	ARM Cortex M3	NXP LPC1837 Xplorer board
lpc43xx	ARM Cortex M4/M0	Hitex Evaluation Board LPC4357
msp430	Texas Instruments MSP430	Texas Instruments EZ430-Chronos
pic24	Microchip PIC24FJ256GA106	openPicus flyportPRO
atmega128	Atmel ATmega128	Ethernut 2

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.

There is also an optional field in the `pico_device` structure, but it is highly recommended that you implement this.

- The callback **link_state** - a pointer to a function that returns info on the link state of the device. When the link is down, a 0 is returned, if it is up, a 1 is returned. Certain protocols depend on this function for hotplug detection, to restart a probing sequence, most notably link-local address allocation. It is expected that mDNS and DNS-SD will also start to rely on this.

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. The given IP address argument must be in network order (i.e. 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 for the link associated to the specified destination. This function will use the currently configured routing table to identify the link that would be used to transmit any traffic directed to the given IP address.

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'. A device may have more than one link configured, i.e. to access multiple networks on the same link.

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 link associated to the local device that was previously configured, corresponding to the 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 associated to the local 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. If zeroed, the call assumes the meaning of adding a default gateway.

- `gateway` - Gateway of the address network. If zeroed, no gateway will be associated to this route, and the traffic towards the destination will be simply forwarded towards the given device.
- `metric` - Metric for this route.
- `link` - Local device interface. If a valid gateway is specified, this parameter is not mandatory, otherwise `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 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 a positive number, which is the ID of the ping operation just started. On error, -1 is returned and pico_err is set appropriately.

Errors

- PICO_ERR_EINVAL - invalid argument
- PICO_ERR_ENOMEM - not enough space

Example

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

3.1.16 pico_icmp4_ping_abort

Description

This function aborts an ongoing ping operation that has previously started using pico_icmp4_ping().

Function prototype

```
int pico_icmp4_ping_abort(int id);
```

Parameters

- id - identification number for the ping operation. This has been returned by pico_icmp4_ping() and it is intended to distinguish the operation to be cancelled.

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_icmp4_ping_abort(id);
```

3.2 IPv6 functions

3.2.1 pico_ipv6_to_string

Description

Convert the internet host address IP to a string in IPv6 colon:hex notation. The result is stored in the char array that ipbuf points to.

Function prototype

```
int pico_ipv6_to_string(char *ipbuf, const uint8_t ip[PICO_SIZE_IP6]);
```

Parameters

- `ipbuf` - Char array to store the result in.
- `ip` - Internet host address in unsigned byte array notation of lenght 16.

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_ipv6_to_string(buf, ip);
```

3.2.2 pico_string_to_ipv6

Description

Convert the IPv6 colon:hex notation into binary form. The result is stored in the `int` that IP points to. The address supplied in `ipstr` can have one of the default forms for IPv6 address description, including at most one abbreviation skipping zeroed fields using "::"

Function prototype

```
int pico_string_to_ipv6(const char *ipstr, uint8_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_ipv6("fe80::1", *ip);
```

3.2.3 pico_ipv6_is_unicast

Description

Check if the provided address is unicast or multicast.

Function prototype

```
int pico_ipv6_is_unicast(struct pico_ip6 *a);
```

Parameters

- `address` - Internet host address.

Return value

Returns 1 if unicast, 0 if multicast.

Example

```
ret = pico_ipv6_is_unicast(address);
```

3.2.4 pico_ipv6_is_multicast

Description

Check if the provided address is a valid Internet multicast address, i.e. it belongs to the range ff00::/8.

Function prototype

```
int pico_ipv6_is_multicast(struct pico_ip6 *a);
```

Parameters

- `address` - Internet host address.

Return value

Returns 1 if a multicast Internet address has been provided.

Example

```
ret = pico_ipv6_is_multicast(address);
```

3.2.5 pico_ipv6_is_global

Description

Check if the provided address is a valid Internet global address, i.e. it belongs to the range 2000::/3.

Function prototype

```
int pico_ipv6_is_global(struct pico_ip6 *a);
```

Parameters

- `address` - Internet host address.

Return value

Returns 1 if a global Internet address has been provided.

Example

```
ret = pico_ipv6_is_global(address);
```

3.2.6 pico_ipv6_is_uniquelocal

Description

Check if the provided address is a valid Internet uniquelocal address, i.e. it belongs to the range fc00::/7.

Function prototype

```
int pico_ipv6_is_uniquelocal(struct pico_ip6 *a);
```

Parameters

- `address` - Internet host address.

Return value

Returns 1 if a uniquelocal Internet address has been provided.

Example

```
ret = pico_ipv6_is_uniquelocal(address);
```

3.2.7 pico_ipv6_is_sitelocal

Description

Check if the provided address is a valid Internet sitelocal address, i.e. it belongs to the range fec0::/10.

Function prototype

```
int pico_ipv6_is_sitelocal(struct pico_ip6 *a);
```

Parameters

- `address` - Internet host address.

Return value

Returns 1 if a sitelocal Internet address has been provided.

Example

```
ret = pico_ipv6_is_sitelocal(address);
```

3.2.8 pico_ipv6_is_linklocal

Description

Check if the provided address is a valid Internet linklocal address, i.e. it belongs to the range fe80::/10.

Function prototype

```
int pico_ipv6_is_linklocal(struct pico_ip6 *a);
```

Parameters

- `address` - Internet host address.

Return value

Returns 1 if a linklocal Internet address has been provided.

Example

```
ret = pico_ipv6_is_linklocal(address);
```

3.2.9 pico_ipv6_is_localhost

Description

Check if the provided address is a valid Internet localhost address, i.e. it is "::1".

Function prototype

```
int pico_ipv6_is_localhost(struct pico_ip6 *a);
```

Parameters

- `address` - Internet host address.

Return value

Returns 1 if a localhost Internet address has been provided.

Example

```
ret = pico_ipv6_is_localhost(address);
```

3.2.10 pico_ipv6_is_undefined

Description

Check if the provided address is a valid Internet undefined address, i.e. it is "::0".

Function prototype

```
int pico_ipv6_is_undefined(struct pico_ip6 *a);
```

Parameters

- `address` - Internet host address.

Return value

Returns 1 if the Internet address provided describes ANY host.

Example

```
ret = pico_ipv6_is_undefined(address);
```

3.2.11 pico_ipv6_source_find

Description

Find the source IP for the link associated to the specified destination. This function will use the currently configured routing table to identify the link that would be used to transmit any traffic directed to the given IP address.

Function prototype

```
struct pico_ip6 *pico_ipv6_source_find(struct pico_ip6 *dst);
```

Parameters

- `address` - Pointer to the destination internet host address as `struct pico_ip6`.

Return value

On success, this call returns the source IP as `struct pico_ip6`. 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_ipv6_source_find(dst);
```

3.2.12 pico_ipv6_link_add

Description

Add a new local device dev inteface, f.e. eth0, with IP address 'address' and netmask 'netmask'. A device may have more than one link configured, i.e. to access multiple networks on the same link.

Function prototype

```
int pico_ipv6_link_add(struct pico_device *dev, struct pico_ip6 address,
struct pico_ip6 netmask);
```

Parameters

- `dev` - Local device.
- `address` - Pointer to the internet host address as `struct pico_ip6`.
- `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_ipv6_link_add(dev, address, netmask);
```

3.2.13 pico_ipv6_link_del

Description

Remove the link associated to the local device that was previously configured, corresponding to the IP address 'address'.

Function prototype

```
int pico_ipv6_link_del(struct pico_device *dev, struct pico_ip6 address);
```

Parameters

- `dev` - Local device.
- `address` - Pointer to the internet host address as `struct pico_ip6`.

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_ipv6_link_del(dev, address);
```

3.2.14 pico_ipv6_link_find

Description

Find the local device associated to the local IP address 'address'.

Function prototype

```
struct pico_device *pico_ipv6_link_find(struct pico_ip6 *address);
```

Parameters

- `address` - Pointer to the internet host address as `struct pico_ip6`.

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_ipv6_link_find(address);
```

3.2.15 pico_ipv6_route_add

Description

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

Function prototype

```
int pico_ipv6_route_add(struct pico_ip6 address, struct pico_ip6 netmask,
struct pico_ip6 gateway, int metric, struct pico_ipv6_link *link);
```

Parameters

- `address` - Pointer to the destination internet host address as `struct pico_ip6`.
- `netmask` - Netmask of the address. If zeroed, the call assumes the meaning of adding a default gateway.
- `gateway` - Gateway of the address network. If zeroed, no gateway will be associated to this route, and the traffic towards the destination will be simply forwarded towards the given device.
- `metric` - Metric for this route.
- `link` - Local device interface. If a valid gateway is specified, this parameter is not mandatory, otherwise `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_ipv6_route_add(dst, netmask, gateway, metric, link);
```

3.2.16 pico_ipv6_route_del

Description

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

Function prototype

```
int pico_ipv6_route_del(struct pico_ip6 address, struct pico_ip6 netmask,
struct pico_ip6 gateway, int metric, struct pico_ipv6_link *link);
```

Parameters

- `address` - Pointer to the destination internet host address as struct `pico_ip6`.
- `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_ipv6_route_del(dst, netmask, gateway, metric, link);
```

3.2.17 pico_ipv6_route_get_gateway

Description

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

Function prototype

```
struct pico_ip6 pico_ipv6_route_get_gateway(struct pico_ip6 *addr)
```

Parameters

- `address` - Pointer to the destination internet host address as struct `pico_ip6`.

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_ip6 pico_ipv6_route_get_gateway(&dest_addr)
```

3.2.18 pico_ipv6_dev_routing_enable

Description

Enable IPv6 Routing messages through the specified interface. On a picoTCP IPv6 machine, when routing is enabled, all possible routes to other links are advertised to the target interfaces. This allows the hosts connected to the target interface to use the picoTCP IPv6 machine as a router towards public IPv6 addresses configured on other interfaces, or reachable through known gateways.

Function prototype

```
struct pico_ip6 pico_ipv6_dev_routing_enable(struct pico_device *dev)
```

Parameters

- `dev` - Pointer to the target device struct `pico_device`.

Return value

On success, zero is returned. On error, -1 is returned and `pico_err` is set appropriately.

Example

```
retval = pico_ipv6_dev_routing_enable(eth1);
```

3.2.19 pico_ipv6_dev_routing_disable

Description

Enable IPv6 Routing messages through the specified interface. On a picoTCP IPv6 machine, when routing is enabled, all possible routes to other links are advertised to the target interface. This function will stop advertising reachable routes to public IPv6 addresses configured on other interfaces, or reachable through known gateways.

Function prototype

```
struct pico_ip6 pico_ipv6_dev_routing_disable(struct pico_device *dev)
```

Parameters

- `dev` - Pointer to the target device struct `pico_device`.

Return value

On success, zero is returned. On error, -1 is returned and `pico_err` is set appropriately.

Example

```
retval = pico_ipv6_dev_routing_disable(eth1);
```

3.3 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.

3.3.1 pico_socket_open

Description

This function will be called to open a socket from the application level. The created socket will be unbound and not connected.

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` - triggered when new data arrives on the socket. A new receive action can be taken by the socket owner because this event indicates there is new data to receive.
- `PICO_SOCK_EV_WR` - triggered when ready to write to the socket. Issuing a write/send call will now succeed if the buffer has enough space to allocate new outstanding data.
- `PICO_SOCK_EV_CONN` - triggered when connection is established (TCP only). This event is received either after a successful call to `pico_socket_connect` to indicate that the connection has been established, or on a listening socket, indicating that a call to `pico_socket_accept` may now be issued in order to accept the incoming connection from a remote host.
- `PICO_SOCK_EV_CLOSE` - triggered when a FIN segment is received (TCP only). This event indicates that the other endpont has closed the connection, so the local TCP layer is only allowed to send new data until a local shutdown or close is initiated. PicoTCP is able to keep the connection half-open (only for sending) after the FIN packet has been received, allowing new data to be sent in the TCP CLOSE_WAIT state.
- `PICO_SOCK_EV_FIN` - triggered when the socket is closed. No further communication is possible from this point on the socket.
- `PICO_SOCK_EV_ERR` - triggered 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);
```

3.3.2 pico_socket_read

Description

This function will be called to read data from a connected socket. The function checks that the socket is bound and connected before attempting to receive data.

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 the buffer where the received data will be stored
- `len` - Length of the buffer (in bytes), represents the maximum amount of bytes that can be read

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);
```

3.3.3 pico_socket_write

Description

This function will be called to write the content of a buffer to a socket that has been previously connected. This function checks that the socket is bound, connected and that it is allowed to send data, i.e. there hasn't been a local shutdown. This is the preferred function to use when writing data from the application to a connected stream.

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 (constant) buffer where the data is stored
- `len` - Length of the data buffer `buf`

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);
```

3.3.4 pico_socket_sendto

Description

This function sends data from the local address to the remote address, without checking whether the remote endpoint is connected. Specifying the destination is particularly useful while sending single datagrams to different destinations upon consecutive calls. This is the preferred mechanism to send datagrams to a remote destination using a UDP socket.

Function prototype

```
int pico_socket_sendto(struct pico_socket *s, const 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 the buffer
- `len` - Length of the buffer `buf`
- `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);
```

3.3.5 pico_socket_recvfrom

Description

This function is called to receive data from the specified socket. It is useful when called in the context of a non-connected socket, to receive the information regarding the origin of the data, namely the origin address and the remote port number.

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 the buffer
- `len` - Maximum allowed length for the data to be stored in the buffer `buf`
- `orig` - Pointer to the origin of the IPv4/IPv6 frame header, can be NULL
- `remote_port` - Pointer to the port number of the sender socket, can be NULL

Return value

On success, this call returns an integer representing the number of bytes read from the socket. On success, if `orig` is not NULL, The address of the remote endpoint is stored in the memory area pointed by `orig`. In the same way, `remote_port` will contain the portnumber of the sending socket, unless a NULL is passed from the caller.

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);
```

3.3.6 Extended Socket operations

The interface provided by sendto/recvfrom can be extended to include more information about the network communication. This is especially useful in UDP communication, and whenever extended information is needed about the single datagram and its encapsulation in the networking layer.

PicoTCP offers an extra structure that can be used to set and retrieve message information while transmitting and receiving datagrams, respectively. The structure `pico_msginfo` is defined as follows:

```
struct pico_msginfo {
    struct pico_device *dev;
    uint8_t ttl;
    uint8_t tos;
};
```

3.3.7 pico_socket_sendto_extended

Description

This function is an extension of the `pico_socket_sendto` function described above. It's exactly the same but it adds up an additional argument to set TTL and QOS information on the outgoing packet which contains the datagram.

The usage of the extended argument makes sense in UDP context only, as the information is set at packet level, and only with UDP there is a 1:1 correspondence between datagrams and IP packets.

Function prototype

```
int pico_socket_sendto_extended(struct pico_socket *s, const void *buf, int len,
void *dst, uint16_t remote_port, struct pico_msginfo *info);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `buf` - Void pointer to the start of the buffer
- `len` - Length of the data that is stored in the buffer (in bytes)
- `dst` - Pointer to the origin of the IPv4/IPv6 frame header
- `remote_port` - Port number of the receiving socket at the remote endpoint
- `info` - Extended information about the packet containing this datagram. Only the fields "ttl" and "tos" are taken into consideeration, while "dev" is ignored.

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

```
struct pico_msginfo info = { };
info.ttl = 5;
bytesWritten = pico_socket_sendto_extended(sk_tcp, buf, len, &sk_tcp->remote_addr,
sk_tcp->remote_port, &info);
```

3.3.8 pico_socket_recvfrom_extended

Description

This function is an extension to the normal `pico_socket_recvfrom` function, which allows to retrieve additional information about the networking layer that has been involved in the delivery of the datagram.

Function prototype

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

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `buf` - Void pointer to the start of the buffer
- `len` - Maximum allowed length for the data to be stored in the buffer `buf`
- `orig` - Pointer to the origin of the IPv4/IPv6 frame header, can be NULL
- `remote_port` - Pointer to the port number of the sender socket, can be NULL
- `info` - Extended information about the incoming packet containing this datagram. The device where the packet was received is pointed by info-¿dev, the maximum TTL for the packet is stored in info-¿ttl, and finally the field info-¿tos keeps track of the flags in IP header's QoS.

Return value

On success, this call returns an integer representing the number of bytes read from the socket. On success, if `orig` is not NULL, The address of the remote endpoint is stored in the memory area pointed by `orig`. In the same way, `remote_port` will contain the portnumber of the sending socket, unless a NULL is passed from the caller.

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

```
struct pico_msginfo info;
bytesRcvd = pico_socket_recvfrom_extended(sk_tcp, buf, bufLen, &peer, &port, &info);
if (info && info->dev) {
    printf("Socket received a datagram via device %s, ttl:%d, tos: %08x\n",
        info->dev->name, info->ttl, info->tos);
}
```

3.3.9 pico_socket_send

Description

This function is called to send data to the specified socket. It checks if the socket is connected and then calls the `pico_socket_sendto` function.

Function prototype

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

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `buf` - Void pointer to the start of the buffer
- `len` - Length of the buffer `buf`

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);
```

3.3.10 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 the buffer
- `len` - Maximum allowed length for the data to be stored in the buffer `buf`

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);
```

3.3.11 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);
```

3.3.12 pico_socket_getname

Description

This function returns the local IP-address and port previously bound to the specified socket.

Function prototype

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

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `local_addr` - Address (IPv4 or IPv6) previously associated to this socket
- `port` - Local portnumber associated to the socket
- `proto` - Proto of the address returned in the `local_addr` field. Can be either `PICO_PROTO_IPV4` or `PICO_PROTO_IPV6`

Return value

On success, this call returns 0 and populates the three fields local_addr `port` and `proto` accordingly. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument(s) provided

Example

```
errMsg = pico_socket_getname(sk_tcp, address, &port, &proto);
if (errMsg == 0) {
    if (proto == PICO_PROTO_IPV4)
        addr4 = (struct pico_ip4 *)address;
    else
        addr6 = (struct pico_ip6 *)address;
}
```

3.3.13 pico_socket_getpeername

Description

This function returns the IP-address of the remote peer connected to the specified socket.

Function prototype

```
int pico_socket_getpeername(struct pico_socket *s, void *remote_addr, uint16_t *port,
                            uint16_t *proto);
```

Parameters

- `s` - Pointer to socket of type `struct pico_socket`
- `remote_addr` - Address (IPv4 or IPv6) associated to the socket remote endpoint
- `port` - Local portnumber associated to the socket
- `proto` - Proto of the address returned in the `local_addr` field. Can be either `PICO_PROTO_IPV4` or `PICO_PROTO_IPV6`

Return value

On success, this call returns 0 and populates the three fields local_addr `port` and `proto` accordingly. On error, -1 is returned, and `pico_err` is set appropriately.

Errors

- `PICO_ERR_EINVAL` - invalid argument(s) provided
- `PICO_ERR_ENOTCONN` - the socket is not connected to any peer

Example

```
errMsg = pico_socket_getpeername(sk_tcp, address, &port, &proto);
if (errMsg == 0) {
    if (proto == PICO_PROTO_IPV4)
        addr4 = (struct pico_ip4 *)address;
    else
        addr6 = (struct pico_ip6 *)address;
}
```

3.3.14 pico_socket_connect

Description

This function connects a local socket to a remote socket of a server that is listening, or permanently associate a remote UDP peer as default receiver for any further outgoing traffic through this socket.

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);
```

3.3.15 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);
```

3.3.16 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);
```

3.3.17 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);
```

3.3.18 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);
```

3.3.19 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/enables the Nagle algorithm (TCP Only).
- `PICO_SOCKET_OPT_KEEPCNT` - Set number of probes for TCP keepalive
- `PICO_SOCKET_OPT_KEEPIDLE` - Set timeout value for TCP keepalive probes (in ms)
- `PICO_SOCKET_OPT_KEEPINTVL` - Set interval between TCP keepalive retries in case of no reply (in ms)
- `PICO_SOCKET_OPT_LINGER` - Set linger time for TCP TIME_WAIT state (in ms)
- `PICO_SOCKET_OPT_RCVBUF` - Set receive buffer size for the socket
- `PICO_SOCKET_OPT_RCVBUF` - Set receive buffer size for the socket
- `PICO_SOCKET_OPT_RCVBUF` - Set receive buffer size for the socket
- `PICO_SOCKET_OPT_SNDBUF` - Set send buffer size for the socket
- `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 in the *pico_ip_mreq* structure passed in the value argument

- PICO_IP_DROP_MEMBERSHIP - Leave the multicast group specified in the *pico_ip_mreq* structure passed in the value argument
- PICO_IP_ADD_SOURCE_MEMBERSHIP - Join the source-specific multicast group specified in the *pico_ip_mreq_source* structure passed in the value argument
- PICO_IP_DROP_SOURCE_MEMBERSHIP - Leave the source-specific multicast group specified in the *pico_ip_mreq_source* structure passed in the value argument

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)
```

3.3.20 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_SOCKET_OPT_RCVBUF - Read current receive buffer size for the socket
- PICO_SOCKET_OPT_SNDBUF - Read current receive buffer size for the socket
- 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);
```

3.4 DHCP client

A DHCP client for obtaining a dynamic IP address. DHCP is supported on multiple interfaces.

3.4.1 pico_dhcp_initiate_negotiation

Description

Initiate a DHCP negotiation. The user passes a callback-function, which will be executed on DHCP success or failure.

Function prototype

```
int pico_dhcp_initiate_negotiation(struct pico_device *device,
  void (*callback)(void *cli, int code), uint32_t *xid);
```

Parameters

- device - the device on which a negotiation should be started.
- callback - the function which is executed on success or failure. This function can be called multiple times. F.e.: initially DHCP succeeded, then the DHCP server is removed long enough from the network for the lease to expire, later the server is added again to the network. The callback is called 3 times: first with code PICO_DHCP_SUCCESS, then with PICO_DHCP_RESET and finally with PICO_DHCP_SUCCESS. The callback may be called before pico_dhcp_initiate_negotiation has returned, f.e. in case of failure to open a socket. The callback has two parameters:

  - `cli` - the identifier of the negotiation
  - `code` - the id indicating success or failure, see further
- `xid` - transaction id of the negotiation. Is set on `PICO_DHCP_SUCCESS`, 0 otherwise.

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

Returns 0 on success, -1 otherwise.

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

```
pico_dhcp_initiate_negotiation(dev, &callback_dhcpclient, &xid);
```

3.4.2 pico_dhcp_client_abort

Description

Cancel the ongoing negotiation. To be used if the operation of obtaining an IP address from a remote DHCP server needs to be aborted, before the callback has been triggered.

Function prototype

```
struct pico_ip4 pico_dhcp_client_abort(uint32_t xid);
```

Parameters

- `xid` - the transaction id returned from the call `pico_dhcp_initiate_negotiation`.

Return value

Returns 0 on success, -1 otherwise (i.e. the XID could not be found in the list of ongoing transactions).

3.4.3 pico_dhcp_get_identifier

Description

Get the identifier needed to pass to all other `pico_dhcp` functions. This function should only be called after a callback occurred with code `PICO_DHCP_SUCCESS`.

Function prototype

```
void *pico_dhcp_get_identifier(uint32_t xid);
```

Parameters

- `xid` - transaction id of the negotiation.

Return value

`void *` - pointer to the identifier.

Example

```
void *cli = pico_dhcp_get_identifier(xid);
```

3.4.4 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 identifier that was provided by the callback on `PICO_DHCP_SUCCESS`.

Return value

`struct pico_ip4` - the address that was assigned.

Example

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

3.4.5 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 identifier that was provided by the callback on `PICO_DHCP_SUCCESS`.

Return value

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

3.4.6 pico_dhcp_get_nameserver

Description

Get the address of the first or the second nameserver 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_nameserver(void *cli, int index);
```

Parameters

- `cli` - the identifier that was provided by the callback on `PICO_DHCP_SUCCESS`.
- `index` - the indes of the domain name server received. Can be either "0" or "1".

Return value

- `struct pico_ip4` - the address of the nameserver that should be used. On failure, e.g. an invalid index was passed, it returns "255.255.255.255". If the IP address of the DNS has not been set, it may return INADDR_ANY.

Example

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

3.5 DHCP server

3.5.1 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_ip4 my_ip` - the IP address of the device performing DHCP. Only IPs of this network will be served.
  - `uint32_t pool_start` - the lowest host number that may be assigned, defaults to 100 if not provided.
  - `uint32_t pool_end` - the highest host number that may be assigned, defaults to 254 if not provided.
  - `uint32_t lease_time` - the advertised lease time in seconds, defaults to 120 if not provided.

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

3.5.2 pico_dhcp_server_destroy

Description

This function stops a previously started DHCP server on the given device.

Function prototype

```
int pico_dhcp_server_destroy(struct pico_device *dev);
```

Parameters

- `dev` - a pointer to a struct `pico_device`, to identify a previously started DHCP server that must be terminated.

Return value

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

Errors

- PICO_ERR_ENOENT - there was no DHCP server running on the given device.

Example

```
struct pico_dhcpd_settings s = { };

s.my_ip.addr = long_be(0x0a280001); /* 10.40.0.1 */

pico_dhcp_server_initiate(&s);
```

3.6 DNS client

3.6.1 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);
```

3.6.2 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, void *arg),
  void *arg);
```

Parameters

- `url` - Pointer to text string containing url text string (e.g. www.google.com).
- `callback` - Callback function, returning the internet host address IP and the provided argument. The returned string has to be freed by the user.
- `arg` - Pointer to an identifier for the request. The pointer is returned in the callback.

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, &identifier);
```

3.6.3 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, void *arg),
  void *arg);
```

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.
- `arg` - Pointer to an identifier for the request. The pointer is returned in the callback.

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, &identifier);
```

3.7 MDNS client

This module can register DNS resource records on the network via Multicast DNS as either ***shared*** or ***unique*** records. Unique records are, as the name implies, unique on the network (the record-name and -type combination is unique) and one single host has claimed the ownership of them. Shared records are records that are not unique on the network, which means multiple hosts can register records with the same record-name and -type combination. For more information on shared and unique resource record sets, see RFC6762.

Unique records are, as it should, defended when somebody else tries to claim the same unique records. When hosts detect such a defense of another host while registering their own records, the conflict will be resolved by choosing another name for the records and another attempt is made to register those new records.

This module only supplies the mechanisms of record registration and resolving on the network, it doesn't parses the contents of them, that's up to the application.

3.7.1 pico_mdns_init

Description

Initialises the entire mDNS-module and sets the hostname for this machine. Sets up the global mDNS socket properly and calls callback when succeeded. Only when the module is properly

initialised, records can be registered on the network.

Function prototype

```
int pico_mdns_init( const char *hostname,
                    struct pico_ip4 address,
                    void (*callback)(pico_mdns_rtree *, char *, void *),
                    void *arg );
```

Parameters

- `hostname` - Hostname to register for this machine. Should end with ‘.local’
- `address` - IPv4-address of this machines interface to generate a hostname record from.
- `cb_initialised` - Callback-function that is called when the initialisation process is done. This will also get called when asynchronous conflicts occur for successfully registered records during run-time. The mDNS-record tree contains the registered records, the char-buffer contains the registered hostname and the void-pointer contains the passed argument.
- `arg` - Argument for callback supplied by user. This can be used if you want to pass some variable into your callback function.

Return value

Returns 0 when the module is properly initialised and the host started registering the hostname. Returns something else went the host failed initialising the module or registering the hostname. `pico_err` is set appropriately.

Errors

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

Example

```
pico_mdns_init("host.local", address, &mdns_init_callback, NULL);
```

3.7.2 pico_mdns_get_hostname

Description

Get the current hostname for this machine.

Function prototype

```
const char * pico_mdns_get_hostname( void );
```

Return value

Returns the current hostname for this machine when the module is initialised, returns NULL when the module is not initialised.

Errors

- `PICO_ERR_EINVAL` - invalid argument

Example

```
char *url = pico_mdns_get_hostname();
```

3.7.3 pico_mdns_set_hostname

Description

Tries to claim a hostname for this machine. Claims automatically a unique A record with the IPv4-address of this host. The hostname won't be set directly when this functions returns, but only if the claiming of the unique record succeeded. Init-callback specified when initialising the module will be called when the hostname-record is successfully registered.

Function prototype

```
int pico_mdns_tryclaim_hostname( const char *url, void *arg );
```

Parameters

- `url` - URL to set the hostname to. Should end with ‚local‘
- `arg` - Argument for init-callback supplied by user. This can be used if you want to pass some variable into your callback function.

Return value

Returns 0 when the host started registering the hostname-record successfully, returns something else when it didn't succeed. `pico_err` is set appropriately.

Errors

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

Example

```
int ret = pico_mdns_tryclaim_hostname("device.local", NULL);
```

3.7.4 pico_mdns_claim

Description

Claims all different mDNS records in a tree in a single API-call. All records in the mDNS record-tree are registered in a single new claim-session.

Function prototype

```
int pico_mdns_claim( pico_mdns_rtree record_tree,
                     void (*callback)(pico_mdns_rtree *,  char *, void *),
                     void *arg );
```

Parameters

- `record_tree` - mDNS record-tree with records to register on the network via Multicast DNS. Can contain ***unique records*** as well as ***shared records***. Declare a mDNS record-tree with the macro 'PICO_MDNS_RTREE_DECLARE(name)', which is actually just a pico_tree-struct, with a comparing-function already set. Records can be added with the preprocessor macro 'PICO_MDNS_RTREE_ADD(pico_mdns_rtree *, struct pico_mdns_record *)'. To create mDNS records see 'pico_mdns_record_create'.
- `callback` - Callback function that gets called when ***ALL*** records in the tree are successfully registered on the network. Records in the returned tree can differ from records originally registered due to conflict-resolution and such.

- `arg` - Argument for callback supplied by user. This can be used if you want to pass some variable into your callback function.

Return value

Returns 0 when the host started registering the record successfully, returns something else when it didn't succeed. `pico_err` is set appropriately.

Errors

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

Example

```
...
PICO_MDNS_RTREE_DECLARE(rtree);
PICO_MDNS_RTREE_ADD(&rtree, &record);
int ret = pico_mdns_claim(rtree, &claimed_cb, NULL);
```

3.7.5 pico_mdns_getrecord

Description

API-call to query a record with a certain URL and type. First checks the cache for this record. If no cache-entry is found, a query will be sent on the wire for this record.

Function prototype

```
int pico_mdns_getrecord( const char *url, uint16_t type,
                         void (*callback)(pico_mdns_rtree *, char *, void *),
                         void *arg );
```

Parameters

- `url` - URL of the DNS name to query records for.
- `type` - DNS type of the records to query for on the network.
- `callback` - Callback to call when records are found or answers to the query are received. This functions can get called multiple times when multiple answers are possible (e.g. with shared records). It's up to the application to aggregate all these received answers, this is possible with a static variable of the type pico_mdns_rtree.
- `arg` - Argument for callback supplied by user. This can be used if you want to pass some variable into your callback function.

Return value

Returns 0 when the host started querying for these records successfully or the records are found in the cache. Returns something else when it didn't succeed. `pico_err` is set appropriately.

Errors

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

Example

```
int ret = pico_mdns_getrecord("_ipp._tcp.local", PICO_DNS_TYPE_PTR, &query_cb, NULL);
```

3.7.6 pico_mdns_record_create

Description

Creates a single standalone mDNS resource record with given name, type and data to register on the network.

Function prototype

```
struct pico_mdns_record *pico_mdns_record_create( const char *url,
                                                  void *_rdata,
                                                  uint16_t datalen,
                                                  uint16_t rtype,
                                                  uint32_t rttl,
                                                  uint8_t flags );
```

Parameters

- `url` - DNS resource record name in URL format. Will be converted to DNS name notation format.
- `_rdata` - Memory buffer with data to insert in the resource record. If data of record should contain a DNS name, the name in the databuffer needs to be in URL-format.
- `datalen` - The exact length in bytes of the _rdata-buffer. If data of record should contain a DNS name (f.e. with PICO_DNS_TYPE_PTR), datalen needs to be pico_dns_strlen(_rdata).
- `rtype` - DNS type of the resource record to be.
- `ttl` - TTL of the resource record to be when registered on the network. In seconds.
- `flags` - With this parameter, you can specify a record as either a shared record or a unique record with respectively PICO_MDNS_RECORD_SHARED- or PICO_MDNS_RECORD_UNIQUE-preprocessor defines. Records are by default registered as unique.

Return value

Returns a pointer to the newly created record on success, returns NULL on failure. `pico_err` is set appropriately.

Errors

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

Example

```
pico_ip4 ip = 0;
pico_string_to_ipv4("10.10.0.5", &(ip.addr));
struct pico_mdns_record *record = pico_mdns_record_create("foo.local",
                                                          &(ip.addr),
                                                          PICO_SIZE_IP4,
                                                          PICO_DNS_TYPE_ANY,
                                                          120,
                                                    PICO_MDNS_RECORD_UNIQUE);
```

3.7.7 IS_HOSTNAME_RECORD

Description

The initialisation-callback can get called multiple times during run-time due to *passive conflict detection*. A passive conflict occurs for unique records when a faulty Multicast DNS-responder doesn't apply conflict resolution after an occurred conflict. A passive conflict can also occur when a peer registers a ***shared*** record with the same name and type combination as a ***unique*** record that the local host already successfully registered on the network. Because of that, shared records have priority over unique records, so unfortunately the local host has to apply the conflict resolution-mechanism to it's earlier uniquely verified record. To be able to notify the application of an updated unique record, the callback gets called given in the initialisation-function. But since that callback maybe parses the returned records as the hostname-records and this isn't necessarily the case when a passive conflict occurs, a mechanism is needed to differ hostname-records from other records. This preprocessor-macro allows this.

Function prototype

```
IS_HOSTNAME_RECORD(record)
```

Parameters

- `record` - mDNS resource record

Return value

Returns 1 when this record is a hostname record, returns 0 when it's not or when given pointer is a NULL pointer.

3.8 DNS SD client

With this module DNS-SD services can be registered on the network to allow Zero Configuration Networking on the device. This is merely a small layer on top of Multicast DNS.

3.8.1 pico_dns_sd_init

Description

Just calls pico_mdns_init in its turn to initialise the mDNS-module. See 'pico_mdns_init' for more information.

3.8.2 pico_dns_sd_register_service

Description

Registers the service with a certain name and type on the network via Multicast DNS.

Function prototype

```
int pico_dns_sd_register_service( const char *name,
                                  const char *type,
                                  uint16_t port,
                                  kv_vector *txt_data,
```

```
uint16_t ttl,
void (*callback)(pico_mdns_rtree *,char *,void *),
void *arg);
```

Parameters

- `name` - Instance-name of the service. Use a descriptive name for it but not longer than 63 characters.
- `type` - The type of the service. For all the possible service types see: `http://www.dns-sd.org/servicetypes.html`
- `port` - The portnumber on which the service runs.
- `txt_data` - Pointer to vector with key-value pairs to insert into the TXT record to give additional information about the service. Use the 'PICO_DNS_SD_KV_VECTOR_DECLARE'-macro to declare a vector for key-value-pairs. This vector will be destroyed when the function returns since there's no need in keeping the contents.
- `ttl` - TTL of the service on the network before it needs to be reconfirmed. In seconds.
- `callback` - Callback function that gets called when the service is successfully registered on the network.
- `arg` - Argument for callback supplied by user. This can be used if you want to pass some variable into your callback function.

Return value

Returns 0 when the module successfully started registering the service, something else on failure. `pico_err` is set appropriately.

Errors

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

Example

```
PICO_DNS_SD_KV_VECTOR_DECLARE(dictionary);
pico_dns_sd_register_service("Printer 2nd Floor", "_printer._sub._http._tcp", 80, \\
&dictionary, 240, &reg_cb, NULL);
```

3.8.3 pico_dns_sd_kv_vector_add

Description

Add a key-value pair the a key-value pair vector.

Function prototype

```
int pico_dns_sd_kv_vector_add( kv_vector *vector, char *key, char *value );
```

Parameters

- `vector` - Pointer to vector to add the pair to. Declare a key-value vector with the 'PICO_DNS_SD_KV_VECTOR_DECLARE'-macro.
- `key` - Key of the pair. Cannot be NULL.
- `value` - Value of the pair. can be NULL, empty ("") or filled ("value").

Return value

Returns 0 when the pair is added successfully, something else on failure. `pico_err` is set appropriately.

Errors

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

Example

```
PICO_DNS_SD_KV_VECTOR_DECLARE(dictionary);
pico_dns_sd_kv_vector_add(&dictionary, "pass", "1234");
pico_dns_sd_kv_vector_add(&dictionary, "color", NULL);
```

3.9 SNTP client

This module allows you to sync your device to to a specified (s)ntp server. You can then retreive the time with the pico_sntp_gettimeofday function.

3.9.1 pico_sntp_sync

Description

Function to sync the local time to a given sntp server.

Function prototype

```
int pico_sntp_sync(char *sntp_server, void (*cb_synced)(pico_err_t status));
```

Parameters

- `sntp_server` - String with the sntp server to get the time from
- `cb_synced` - Callback function that is called when the synchronisation process is done. The status variable indicates wheter the synchronisation was succesfull or not.

Return value

On success, this call returns 0 if the synchronisation 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

Example

```
int ret = pico_sntp_sync("ntp.nasa.gov", &callback);
```

3.9.2 pico_sntp_gettimeofday

Description

Function to get the current time. Be sure to call the pico_sntp_sync function to synchronise BEFORE calling this function.

Function prototype

```
int pico_sntp_gettimeofday(struct pico_timeval *tv);
```

Parameters

- `tv` - Pointer to a time_val struct in which the current time will be set.

Return value

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

Example

```
int ret = pico_sntp_gettimeofday(tv);
```

3.10 IGMP

This module allows the user to join and leave ipv4 multicast groups. The module is based on the IGMP version 3 protocol and it's backwards compatible with version 2. Version 1 is not supported. The IGMP module is completly driven from socket calls (3.3.19) and non of the IGMP application interface functions should be called from the user himself. If however, by any reason, it's necessary for the user to do this, the following function call is provided:

3.10.1 pico_igmp_state_change

Description

Change the state of the host to Non-member, Idle member or Delaying member.

Function prototype

```
int pico_igmp_state_change(struct pico_ip4 *mcast_link, struct pico_ip4 *mcast_group,
 uint8_t filter_mode, struct pico_tree *_MCASTFilter, uint8_t state)
```

Parameters

- `mcast_link` - the link on which that multicast group should be joined.
- `mcast_group` - the address of the multicast group you want to join.
- `filter_mode` - the kind of source filtering, if applied.
- `_MCASTFilter` - list of multicast sources on which source filtering might be applied.
- `state` - the prefered new state.

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_ENOMEM - Not enough space
- PICO_ERR_EPROTONOSUPPORT - Invalid protocol (or protocol version) found on the link
- PICO_ERR_EFAULT - Internal error

3.11 MLD

This module allows the user to join and leave ipv6 multicast groups. The module is based on the MLD version 2 protocol and it's backwards compatible with version 1. The MLD module is completly driven from socket calls (3.3.19) and non of the MLD application interface functions should be called from the user himself. If however, by any reason, it's necessary for the user to do this, the following function call is provided:

3.11.1 pico_mld_state_change

Description

Change the state of the host to Non-listener, Idle listener or Delaying listener.

Function prototype

```
int pico_mld_state_change(struct pico_ip6 *mcast_link, struct pico_ip6 *mcast_group,
 uint8_t filter_mode, struct pico_tree *_MCASTFilter, uint8_t state)
```

Parameters

- mcast_link - the link on which that multicast group should be joined.
- mcast_group - the address of the multicast group you want to join.
- filter_mode - the kind of source filtering, if applied.
- _MCASTFilter - list of multicast sources on which source filtering might be applied.
- state - the prefered new state.

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_ENOMEM - Not enough space
- PICO_ERR_EPROTONOSUPPORT - Invalid protocol (or protocol version) found on the link
- PICO_ERR_EFAULT - Internal error

3.12 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.

3.12.1 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 assign on the marked packet
- `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

3.12.2 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);
```

3.13 SLAACV4 Module

3.13.1 pico_slaacv4_claimip

Description

This function starts the ip claiming process for a device. It will generate first the local link ip using as seed the mac address of the device. Then it will start the claim procedure described in RFC3927. In case of success the IP is registered to the IP layer and returned using the callback function. In case of error, code SLAACV4_ERROR is returned. Errors occur when the maximum number of conflicts is reached. Use the IP returned only if the return code is SLAACV4_SUCCESS.

Function prototype

```
pico_slaacv4_claimip(struct pico_device *dev, void (*cb)(struct pico_ip4 *ip, uint8_t
code));
```

Parameters

- `dev` - a pointer to a struct `pico_device`
- `*cb` - a callback function returning the ip claimed and a return code (SLAACV4_ERROR — SLAACV4_SUCCESS)

Return value

0 returned if the claiming has started successfully

Example

```
dev = pico_get_device(sdev);

ret = pico_slaacv4_claimip(dev, slaacv4_cb);
```

3.13.2 pico_slaacv4_unregisterip

Description

This function allows to unregister the local link ip in usage. The function will remove from the route table the local link ip and will reset the internal state of the SLAACV4 module

Function prototype

```
void pico_slaacv4_unregisterip(void);
```

3.14 TFTP

This module provides support for Trivial File Transfer Protocol (TFTP). The support includes client and server implementation, both of them can be active at the same time.

Flows must be split up into TFTP blocks on the sender side, and reassembled from block len on the receiving side. Please note that a block whose size is less than the block size indicates the end of the transfer.

To indicate the end of a transfer where the content is aligned with the block size, an additional transmission of zero bytes must follow the flow.

Function pico_tftp_listen must be used to start the server with a proper callback that should be provided by the user. To reject a request received by the server the server callback must call pico_tftp_reject_request.

In order to start transmission or reception of files a session handler must be obtained with a call to pico_tftp_session_setup. The created session may take advantage of the Extenxed Options of the TFTP protocol invoking pico_tftp_set_option before starting using it.

Real file transaction is started using the functions pico_tftp_start_tx and pico_tftp_start_rx; both require a callback that must be provided by the user to handle single chunks of the transmission. The transmitter callback must use pico_tftp_send function to send each block of data.

In case of problem the session can be aborted (and an error message is sent to the remote side) using pico_tftp_abort.

When a transfer is complete the session became invalid and must not be used any more.

Application driven interface

In some use case is preferable to have an application driven behaviour. The API provide 5 specific functions to use TFTP in this scenario.

The way to obtain a session handler suited for this purpose is an invocation to the function pico_tftp_app_setup. The synchro variable passed to this function will play a key role during the management of the transfer.

As usual the section can be instructed to use Extended Options using pico_tftp_set_option before starting the file transfer.

Once the session is created, the application can start receiving a file with a call to the function pico_tftp_app_start_rx or, if needs to transmit, invoking pico_tftp_app_start_tx.

After the file transfer is started the user is allowed to perform data handling only when the synchro variable associated with the session is not 0. It is set to 0 after calling pico_tftp_app_setup.

A value that differ to 0 means that a single chunk is ready to be handled.

Single chunk of data are received using pico_tftp_get and transmitted with the use of the function pico_tftp_put.

Once the file transfer ends, both for completion or in case of error, the session is no more valid.

3.14.1 pico_tftp_listen

Description

Start up a TFTP server listening for GET/PUT requests on the given port. The function pointer passed as callback in the `cb` argument will be invoked upon a new transfer request received from the network, and the call will pass the information about:

- The address of the remote peer asking for a transfer
- The remote port of the peer
- The type of transfer requested, via the `opcode` parameter being either `PICO_TFTP_RRQ` or `PICO_TFTP_WRQ`, for get or put requests respectively.

Function prototype

```
int pico_tftp_listen(uint16_t family, void (*cb)(union pico_address *addr,
                uint16_t port, uint16_t opcode, char *filename, int32_t len));
```

Parameters

- `family` - The chosen socket family. Accepted values are `PICO_PROTO_IPV4` for IPv4 and `PICO_PROTO_IPV6` for IPv6.
- `cb` - a pointer to the callback function, defined by the user, that will be called upon a new transfer request.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
/* Example of a TFTP listening service callback */

void tftp_listen_cb(union pico_address *addr, uint16_t port,
                    uint16_t opcode, char *filename, int32_t len)
{
    struct note_t *note;
    struct pico_tftp_session *session;

    printf("TFTP listen callback (BASIC) from remote port %" PRIu16 ".\n",
           short_be(port));
    if (opcode == PICO_TFTP_RRQ) {
        printf("Received TFTP get request for %s\n", filename);
        note = transfer_prepare(&session, 't', filename, addr, family);
        start_tx(session, filename, port, cb_tftp_tx, note);
    } else if (opcode == PICO_TFTP_WRQ) {
        printf("Received TFTP put request for %s\n", filename);
```

```
        note = transfer_prepare(&session, 'r', filename, addr, family);
        start_rx(session, filename, port, cb_tftp_rx, note);
    }
}

// Code fragment to demostrate the use of pico_tftp_listen:
if (!is_server_enabled) {
    pico_tftp_listen(PICO_PROTO_IPV4, (commands->operation == 'S') ?
                    tftp_listen_cb_opt : tftp_listen_cb);
    is_server_enabled = 1;
}
```

3.14.2 pico_tftp_reject_request

Description

This message is used in listen callback to reject a request with an error message.

Function prototype

```
int pico_tftp_reject_request(union pico_address *addr, uint16_t port,
                             uint16_t error_code, const char *error_message);
```

Parameters

- `addr` - The address of the remote peer; it must match the address where the request came from.
- `port` - The port on the remote peer; it must match the address where the request came from.
- `error_code` - Error reason, possible values are:

TFTP_ERR_UNDEF	Not defined, see error message (if any)
TFTP_ERR_ENOENT	File not found
TFTP_ERR_EACC	Access violation
TFTP_ERR_EXCEEDED	Disk full or allocation exceeded
TFTP_ERR_EILL	Illegal TFTP operation
TFTP_ERR_ETID	Unknown transfer ID
TFTP_ERR_EEXIST	File already exists
TFTP_ERR_EUSR	No such user
TFTP_ERR_EOPT	Option negotiation

- `message` - Text message to attach.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
void tftp_listen_cb_opt(union pico_address *addr, uint16_t port,
                        uint16_t opcode, char *filename, int32_t len)
```

```
{
    struct note_t *note;
    struct pico_tftp_session *session;
    int options;
    uint8_t timeout;
    int32_t filesize;
    int ret;

    printf("TFTP listen callback (OPTIONS) from remote port %" PRIu16 ".\n",
           short_be(port));
    /* declare the options we want to support */
    ret = pico_tftp_parse_request_args(filename, len, &options,
                                       &timeout, &filesize);
    if (ret)
        pico_tftp_reject_request(addr, port, TFTP_ERR_EOPT,
                                 "Malformed request");

    if (opcode == PICO_TFTP_RRQ) {
        printf("Received TFTP get request for %s\n", filename);
        note = transfer_prepare(&session, 'T', filename, addr, family);

        if (options & PICO_TFTP_OPTION_TIME)
            pico_tftp_set_option(session, PICO_TFTP_OPTION_TIME, timeout);
        if (options & PICO_TFTP_OPTION_FILE) {
            ret = get_filesize(filename);
            if (ret < 0) {
                pico_tftp_reject_request(addr, port, TFTP_ERR_ENOENT,
                                         "File not found");
                return;
            }
            pico_tftp_set_option(session, PICO_TFTP_OPTION_FILE, ret);
        }

        start_tx(session, filename, port, cb_tftp_tx_opt, note);
    } else { /* opcode == PICO_TFTP_WRQ */
        printf("Received TFTP put request for %s\n", filename);

        note = transfer_prepare(&session, 'R', filename, addr, family);
        if (options & PICO_TFTP_OPTION_TIME)
            pico_tftp_set_option(session, PICO_TFTP_OPTION_TIME, timeout);
        if (options & PICO_TFTP_OPTION_FILE)
            pico_tftp_set_option(session, PICO_TFTP_OPTION_FILE, filesize);

        start_rx(session, filename, port, cb_tftp_rx_opt, note);
    }
}
```

3.14.3 pico_tftp_session_setup

Description

Obtain a session handler to use for the next file transfer with a remote location.

Function prototype

```
struct pico_tftp_session * pico_tftp_session_setup(union pico_address *a,
                                                   uint16_t family);
```

Parameters

- `a` - The address of the peer to be contacted. In case of a solicited transfer, it must match the address where the request came from.
- `family` - The chosen socket family. Accepted values are `PICO_PROTO_IPV4` for IPv4 and `PICO_PROTO_IPV6` for IPv6.

Return value

In case of success a session handler is returned. In case of failure, NULL is returned and pico_err is set accordingly.

Example

```
struct pico_tftp_session * make_session_or_die(union pico_address *addr,
                                               uint16_t family)
{
    struct pico_tftp_session * session;

    session = pico_tftp_session_setup(addr, family);
    if (!session) {
        fprintf(stderr, "TFTP: Error in session setup\n");
        exit(3);
    }
    return session;
}
```

3.14.4 pico_tftp_set_option

Description

This function is used to require the use of Extended Options for TFTP transfer associate to a session according to RFC 2347 and RFC 2349. It should be used before the invocation of pico_tftp_start_rx or pico_tftp_start_tx unless the setting is related to the timeout. In order to require Transfer size Option PICO_TFTP_OPTION_FILE must be used and his value set to the file size in case of a Write Request or to 0 in case of a Read Request. To require to adopt a specific fixed value for the timeout PICO_TFTP_OPTION_TIME must be used with a value ranging between 1 and 255. If this option is set to a value of 0 (or not used at all) an adaptive timeout algorithm will take care of the retransmissions.

Function prototype

```
int pico_tftp_set_option(struct pico_tftp_session *session,
```

```
                        uint8_t type, int32_t value);
```

Parameters

- `session` - Section handler to use for the file transfer.
- `type` - Option to set; accepted values are PICO_TFTP_OPTION_FILE for Transfer size Option or PICO_TFTP_OPTION_TIME for Timeout interval Option.
- `value` - Option value to send.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
filesize = get_filesize(commands->filename);
if (filesize < 0) {
    fprintf(stderr, "TFTP: unable to read size of file %s\n",
            commands->filename);
    exit(3);
}
pico_tftp_set_option(session, PICO_TFTP_OPTION_FILE, filesize);
start_tx(session, commands->filename, short_be(PICO_TFTP_PORT),
         cb_tftp_tx_opt, note);
```

3.14.5 pico_tftp_get_option

Description

This function is used to retrieve the values of Extended Options that has been set to a session according to RFC 2347 and RFC 2349. In order to ask Transfer size Option value PICO_TFTP_OPTION_FILE must be used; it may be used for example for example in receiver callback for calculation of remaining bytes to be received to complete the current transfer. To query the timeout PICO_TFTP_OPTION_TIME must be used; a value ranging between 1 and 255 will be returned in the value parameter if the fixed interval is in place. If the call return -1 and pico_err is set to PICO_ERR_ENOENT the adaptive timeout algorithm is running.

Function prototype

```
int pico_tftp_set_option(struct pico_tftp_session *session,
                         uint8_t type, int32_t *value);
```

Parameters

- `session` - Section handler to use for the file transfer.
- `type` - Option to query; accepted values are PICO_TFTP_OPTION_FILE for Transfer size Option or PICO_TFTP_OPTION_TIME for Timeout interval Option.
- `value` - Pointer to an integer variable where to store the value.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
int cb_tftp_tx_opt(struct pico_tftp_session *session, uint16_t event,
                   uint8_t *block, int32_t len, void *arg)
{
    int ret;
    int32_t filesize;

    if (event == PICO_TFTP_EV_OPT) {
        ret = pico_tftp_get_option(session, PICO_TFTP_OPTION_FILE, &filesize);
        if (ret)
            printf("TFTP: Option filesize is not used\n");
        else
            printf("TFTP: We expect to transmit %" PRId32 " bytes\n",
                   filesize);
        event = PICO_TFTP_EV_OK;
    }

    return cb_tftp_tx(session, event, block, len, arg);
}
```

3.14.6 pico_tftp_parse_request_args

Description

This function is used to extract Extension Options eventually present in Read or Write request (in the listen callback) or in Option ACKnowledge messages (in transmitter or receiver callback when event is equal to PICO_TFTP_EV_OPT). Note that timeout and filesize are modified only if the corresponding option is found in the received message.

Function prototype

```
int pico_tftp_parse_request_args(char *args, int32_t len, int *options,
                                 uint8_t *timeout, int32_t *filesize);
```

Parameters

- `args` - Pointer to the buffer containing the arguments: filename for listen callback and block for rx or tx callback.
- `len` - Length of the buffer containing the arguments; same value of the len parameter in callbacks.
- `options` - Pointer to the variable that will contain the set of options found. Presence of single options can be then verified anding it with PICO_TFTP_OPTION_FILE or PICO_TFTP_OPTION_TIME.

- `timeout` - Pointer to the variable that will contain the timeout value (if present in the options).
- `filesize` - Pointer to the variable that will contain the filesize value (if present in the options)..

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
void tftp_listen_cb_opt(union pico_address *addr, uint16_t port,
                        uint16_t opcode, char *filename, int32_t len)
{
    struct note_t *note;
    struct pico_tftp_session *session;
    int options;
    uint8_t timeout;
    int32_t filesize;
    int ret;

    printf("TFTP listen callback (OPTIONS) from remote port %" PRIu16 ".\n",
            short_be(port));
    /* declare the options we want to support */
    ret = pico_tftp_parse_request_args(filename, len, &options,
                                       &timeout, &filesize);
    if (ret)
        pico_tftp_reject_request(addr, port, TFTP_ERR_EOPT,
                                 "Malformed request");

    if (opcode == PICO_TFTP_RRQ) {
        printf("Received TFTP get request for %s\n", filename);
        note = transfer_prepare(&session, 'T', filename, addr, family);

        if (options & PICO_TFTP_OPTION_TIME)
            pico_tftp_set_option(session, PICO_TFTP_OPTION_TIME, timeout);
        if (options & PICO_TFTP_OPTION_FILE) {
            ret = get_filesize(filename);
            if (ret < 0) {
                pico_tftp_reject_request(addr, port, TFTP_ERR_ENOENT,
                "File not found");
                return;
            }
            pico_tftp_set_option(session, PICO_TFTP_OPTION_FILE, ret);
        }

        start_tx(session, filename, port, cb_tftp_tx_opt, note);
    } else { /* opcode == PICO_TFTP_WRQ */
        printf("Received TFTP put request for %s\n", filename);
```

```
        note = transfer_prepare(&session, 'R', filename, addr, family);
        if (options & PICO_TFTP_OPTION_TIME)
            pico_tftp_set_option(session, PICO_TFTP_OPTION_TIME, timeout);
        if (options & PICO_TFTP_OPTION_FILE)
            pico_tftp_set_option(session, PICO_TFTP_OPTION_FILE, filesize);

        start_rx(session, filename, port, cb_tftp_rx_opt, note);
    }
}
```

3.14.7 pico_tftp_start_tx

Description

Start a TFTP transfer. The action can be unsolicited (client PUT operation) or solicited (server responding to a GET request). In either case, the transfer will happen one block at a time, and the callback provided by the user will be called to notify the acknowledgement for the successful of each transfer, transfer of the last block, reception of an option acknowledge message (client mode) or whenever an error occurs. Any error during the TFTP transfer will cancel the transfer itself. The possible values for the `event` variable in callback are:

- `PICO_TFTP_EV_OK` Time to send another chunk of data.
- `PICO_TFTP_EV_OPT` Option acknowledge has been received.
- `PICO_TFTP_EV_ERR_PEER` An error has occurred remotely.
- `PICO_TFTP_EV_ERR_LOCAL` An internal error has occurred.

Function prototype

```
int pico_tftp_start_tx(struct pico_tftp_session *session, uint16_t port,
        const char *filename,
        int (*user_cb)(struct pico_tftp_session *session, uint16_t event,
                       uint8_t *block, int32_t len, void *arg),
        void *arg);
```

Parameters

- `session` - Session handler to use for the file transfer.
- `port` - The port on the remote peer.
- `filename` - The name of the file to be transferred. In case of solicited transfer, it must match the filename provided during the request.
- `user_cb` - The callback provided by the user to be called upon each block transfer, option acknowledge or in case of error.
- `arg` - The pointer is sent as argument to the callback.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
void start_tx(struct pico_tftp_session *session,
    const char *filename, uint16_t port,
    int (*tx_callback)(struct pico_tftp_session *session, uint16_t err,
```

```
                        uint8_t *block, int32_t len, void *arg),
    struct note_t *note)
{
    if (pico_tftp_start_tx(session, port, filename, tx_callback, note)) {
        fprintf(stderr, "TFTP: Error in initialization\n");
        exit(1);
    }
}
```

3.14.8 pico_tftp_send

Description

Send the next block during an active TFTP transfer. This is ideally called every time the user callback is triggered by the protocol, indicating that the transfer of the last block has been acknowledged. The user should not call this function unless it's solicited by the protocol during an active transmit session.

Function prototype

```
int32_t pico_tftp_send(struct pico_tftp_session *session,
                       const uint8_t *data, int32_t len);
```

Parameters

- `session` - the session handler to use for the file transfer.
- `data` - the content of the block to be transferred.
- `len` - the size of the buffer being transmitted. If < `BLOCKSIZE`, the transfer is concluded. In order to terminate a transfer where the content is aligned to `BLOCKSIZE`, a zero-sized `pico_tftp_send` must be called at the end of the transfer.

Return value

In case of success, the number of bytes transmitted is returned. In case of failure, -1 is returned and pico_err is set accordingly.

Example

```
int cb_tftp_tx(struct pico_tftp_session *session, uint16_t event,
               uint8_t *block, int32_t len, void *arg)
{
    struct note_t *note = (struct note_t *) arg;

    if (event != PICO_TFTP_EV_OK) {
        fprintf(stderr, "TFTP: Error %" PRIu16 ": %s\n", event, block);
        exit(1);
    }

    len = read(note->fd, tftp_txbuf, PICO_TFTP_PAYLOAD_SIZE);

    if (len >= 0) {
        note->filesize += len;
        pico_tftp_send(session, tftp_txbuf, len);
```

```
        if (len < PICO_TFTP_PAYLOAD_SIZE) {
            printf("TFTP: file %s (%" PRId32
                   " bytes) TX transfer complete!\n",
                   note->filename, note->filesize);
            close(note->fd);
            del_note(note);
        }
    } else {
        perror("read");
        fprintf(stderr,
                "Filesystem error reading file %s,"
                " cancelling current transfer\n", note->filename);
        pico_tftp_abort(session, TFTP_ERR_EACC, "Error on read");
        del_note(note);
    }

    if (!clipboard)
        pico_timer_add(3000, deferred_exit, NULL);

    return len;
}
```

3.14.9 pico_tftp_start_rx

Description

Start a TFTP transfer. The action can be unsolicited (client GET operation) or solicited (server responding to a PUT request). In either case, the transfer will happen one block at a time, and the callback provided by the user will be called upon successful transfer of a block, whose content can be directly accessed via the `block` field, reception of an option acknowledge messagge (client mode) or whenever an error occurs. The possible values for the `event` variable in callback are:

- `PICO_TFTP_EV_OK` Previously sent block has been acknowledge.
- `PICO_TFTP_EV_OPT` Option acknowledge has been received.
- `PICO_TFTP_EV_ERR_PEER` An error has occurrend remotely.
- `PICO_TFTP_EV_ERR_LOCAL` An internal error has occurred.

Function prototype

```
int pico_tftp_start_rx(struct pico_tftp_session *session, uint16_t port,
        const char *filename,
        int (*user_cb)(struct pico_tftp_session *session, uint16_t event,
                       uint8_t *block, int32_t len, void *arg),
        void *arg);
```

Parameters

- `session` - the session handler to use for the file transfer.
- `port` - The port on the remote peer.
- `filename` - The name of the file to be transfered. In case of solicited transfer, it must match the filename provided during the request.

- `user_cb` - The callback provided by the user to be called upon each block transfer, option acknowledge or in case of error. This is the callback where the incoming data is processed. When len is less than the block size, the transfer is over.
- `arg` - The pointer sent as argument to the callback.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
void start_rx(struct pico_tftp_session *session,
          const char *filename, uint16_t port,
          int (*rx_callback)(struct pico_tftp_session *session, uint16_t err,
                             uint8_t *block, int32_t len, void *arg),
          struct note_t *note)
{
    if (pico_tftp_start_rx(session, port, filename, rx_callback, note)) {
        fprintf(stderr, "TFTP: Error in initialization\n");
        exit(1);
    }
}
```

3.14.10 pico_tftp_get_file_size

Description

This function is used to retrieve the file size (if transmitted by the remote or set as session option). It is equivalent to a call to pico_tftp_get_option(session, PICO_TFTP_OPTION_FILE, &file_size);

Function prototype

```
int pico_tftp_get_file_size(struct pico_tftp_session *session,
                            int32_t *file_size);
```

Parameters

- `session` - Section handler to use for the file transfer.
- `file_size` - Pointer to an integer variable where to store the value.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
ret = pico_tftp_get_file_size(session, &file_size);
if (ret)
    printf("Information about file size has not been received"\n);
```

3.14.11 pico_tftp_abort

Description

When called this function aborts associated ongoing transmission and notifying the other endpoint with a proper error message. After a call to this function the session is closed automatically.

Function prototype

```
int pico_tftp_abort(struct pico_tftp_session *session,
                    uint16_t error, const char *reason);
```

Parameters

- `session` - the session handler related to the session to abort.
- `error` - Error reason code, possible values are:

TFTP_ERR_UNDEF	Not defined, see error message (if any)
TFTP_ERR_ENOENT	File not found
TFTP_ERR_EACC	Access violation
TFTP_ERR_EXCEEDED	Disk full or allocation exceeded
TFTP_ERR_EILL	Illegal TFTP operation
TFTP_ERR_ETID	Unknown transfer ID
TFTP_ERR_EEXIST	File already exists
TFTP_ERR_EUSR	No such user
TFTP_ERR_EOPT	Option negotiation

- `reason` - Text message to attach.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
int cb_tftp_rx(struct pico_tftp_session *session, uint16_t event,
               uint8_t *block, int32_t len, void *arg)
{
    struct note_t *note = (struct note_t *) arg;
    int ret;

    if (event != PICO_TFTP_EV_OK) {
        fprintf(stderr, "TFTP: Error %" PRIu16 ": %s\n", event, block);
        exit(1);
    }

    note->filesize += len;
    if (write(note->fd, block, len) < 0) {
        perror("write");
        fprintf(stderr, "Filesystem error writing file %s,"
                " cancelling current transfer\n", note->filename);
        pico_tftp_abort(session, TFTP_ERR_EACC, "Error on write");
        del_note(note);
```

```
    } else {
        if (len != PICO_TFTP_PAYLOAD_SIZE) {
            printf("TFTP: file %s (%" PRId32
                   " bytes) RX transfer complete!\n",
                   note->filename, note->filesize);
            close(note->fd);
            del_note(note);
        }
    }

    if (!clipboard)
        pico_timer_add(3000, deferred_exit, NULL);

    return len;
}
```

3.14.12 pico_tftp_close_server

Description

This function is used to shutdown the TFTP server.

Function prototype

```
int pico_tftp_close_server(void);
```

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
ret = pico_tftp_close_server();
if (ret)
    printf(stderr, "Failure shutting down the server\n");
```

3.14.13 pico_tftp_app_setup

Description

Obtain a session handler to use for the next file transfer with a remote location in application driven mode.

Function prototype

```
struct pico_tftp_session * pico_tftp_app_setup(union pico_address *a,
        uint16_t port, uint16_t family, int *synchro);
```

Parameters

- `a` - The address of the peer to be contacted. In case of a solicited transfer, it must match the address where the request came from.
- `port` - The port on the remote peer.
- `family` - The chosen socket family. Accepted values are `PICO_PROTO_IPV4` for IPv4 and `PICO_PROTO_IPV6` for IPv6.
- `synchro` - Variable to handle the synchronization.

Return value

In case of success a session handler is returned. In case of failure, NULL is returned and pico_err is set accordingly.

Example

```
session = pico_tftp_app_setup(&server_address, short_be(PICO_TFTP_PORT),
                              PICO_PROTO_IPV4, &synchro);
if (!session) {
    fprintf(stderr, "Error in pico_tftp_app_setup\n");
    exit(1);
}
```

3.14.14 pico_tftp_app_start_rx

Description

Application driven function used to request to read a remote file. The transfer will happen one block at a time using pico_tftp_app_get.

Function prototype

```
int pico_tftp_app_start_rx(struct pico_tftp_session *session,
                           const char *filename);
```

Parameters

- `session` - Session handler to use for the file transfer.
- `filename` - The name of the file to be received.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
printf("Start receiving file %s with options set to %d\n", filename, options);

if (options) {
```

```
    ret = pico_tftp_set_option(session, PICO_TFTP_OPTION_FILE, 0);
    if (ret) {
        fprintf(stderr, "Error in pico_tftp_set_option\n");
        exit(1);
    }
}

ret = pico_tftp_app_start_rx(session, filename);
if (ret) {
    fprintf(stderr, "Error in pico_tftp_app_start_rx\n");
    exit(1);
}
```

3.14.15 pico_tftp_app_start_tx

Description

Application driven function used to request to write a remote file. The transfer will happen one block at a time using pico_tftp_app_put.

Function prototype

```
int pico_tftp_app_start_tx(struct pico_tftp_session *session,
                           const char *filename);
```

Parameters

- `session` - Session handler to use for the file transfer.
- `filename` - The name of the file to be sent.

Return value

This function returns 0 if succeeds or -1 in case of errors (pico_err is set accordingly).

Example

```
printf("Start sending file %s with options set to %d\n", filename, options);

if (options) {
    ret = get_filesize(filename);
    if (ret < 0) {
        fprintf(stderr, "Error in get_filesize\n");
        exit(1);
    }

    ret = pico_tftp_set_option(session, PICO_TFTP_OPTION_FILE, ret);
    if (ret) {
        fprintf(stderr, "Error in pico_tftp_set_option\n");
        exit(1);
```

```
    }
}

ret = pico_tftp_app_start_tx(session, filename);
if (ret) {
    fprintf(stderr, "Error in pico_tftp_app_start_rx\n");
    exit(1);
}
```

3.14.16 pico_tftp_get

Description

Read the next block during an active TFTP transfer. The len field must always be equal to PICO_TFTP_PAYLOAD_SIZE. Once the file has been sent or after an error the session is no more valid.

Function prototype

```
int32_t pico_tftp_get(struct pico_tftp_session *session,
                      uint8_t *data, int32_t len);
```

Parameters

- `session` - Session handler to use for the file transfer.
- `data` - Buffer where to store the acquired payload.
- `len` - Length of the buffer size to receive; it is equal to the fixed chunk size.

Return value

This function returns the number of received bytes of payload (0 included) if succeeds. In case of error a negative number is returned.

Errors

- `-1` At least one of the passed arguments are invalid.
- `-PICO_TFTP_EV_ERR_PEER` Remote failure.
- `-PICO_TFTP_EV_ERR_LOCAL` Local failure.

Example

```
for(;left; left -= countdown) {
    usleep(2000); //PICO_IDLE();
    pico_stack_tick();
    if (countdown)
        continue;

    if (*synchro) {
        len = pico_tftp_get(session, buf, PICO_TFTP_PAYLOAD_SIZE);
        if (len < 0) {
```

```
            fprintf(stderr, "Failure in pico_tftp_get\n");
            close(fd);
            countdown = 1;
            continue;
        }
        ret = write(fd, buf, len);
        if (ret < 0) {
            fprintf(stderr, "Error in write\n");
            pico_tftp_abort(session, TFTP_ERR_EXCEEDED, "File write error");
            close(fd);
            countdown = 1;
            continue;
        }
        printf("Written %" PRId32 " bytes to file (synchro=%d)\n",
               len, *synchro);

        if (len != PICO_TFTP_PAYLOAD_SIZE) {
            close(fd);
            printf("Transfer complete!\n");
            countdown = 1;
        }
    }
}
```

3.14.17 pico_tftp_put

Description

Send the next block during an active TFTP transfer. The len field, with the exception of last invocation must always be equal to PICO_TFTP_PAYLOAD_SIZE. Once the file has been sent or after an error the session is no more valid.

Function prototype

```
int32_t pico_tftp_put(struct pico_tftp_session *session,
                      uint8_t *data, int32_t len);
```

Parameters

- `session` - Session handler to use for the file transfer.
- `data` - Pointer to the data to be transmitted.
- `len` - Length of the buffer size to transmit; last chunk must be < of the maximum buffer size (0 if file size was a multiple of maximum buffer size).

Return value

This function returns the number of transmitted payload data (len) if succeeds. In case of error a negative number is returned.

Errors

- `-1` At least one of the passed arguments are invalid.
- `-PICO_TFTP_EV_ERR_PEER` Remote failure.
- `-PICO_TFTP_EV_ERR_LOCAL` Local failure.

Example

```
for(;left; left -= countdown) {
    usleep(2000); //PICO_IDLE();
    pico_stack_tick();
    if (countdown)
        continue;

    if (*synchro) {
        ret = read(fd, buf, PICO_TFTP_PAYLOAD_SIZE);
        if (ret < 0) {
            fprintf(stderr, "Error in read\n");
            pico_tftp_abort(session, TFTP_ERR_EACC, "File read error");
            close(fd);
            countdown = 1;
            continue;
        }
        printf("Read %" PRId32 " bytes from file (synchro=%d)\n",
               len, *synchro);

        len = pico_tftp_put(session, buf, ret);
        if (len < 0) {
            fprintf(stderr, "Failure in pico_tftp_put\n");
            close(fd);
            countdown = 1;
            continue;
        }

        if (len != PICO_TFTP_PAYLOAD_SIZE) {
            close(fd);
            printf("Transfer complete!\n");
            countdown = 1;
        }
    }
}
```

3.15 Point-to-Point Protocol (PPP)

PPP consists in a family of data-link protocols, providing link control, configuration and authentication over a point-to-point link. In a connected embedded system, it is often used to access dial-up modems over serial lines.

This module supports GSM modem configuration by implementing part of ETSI TS 127 007.

From the picoTCP perspective, each PPP capable device may be abstracted into its own instance that can be created using `pico_ppp_create`.

Any GSM/GPRS/3G/HSDPA module, exporting a non-blocking serial interface, such as SPI or UART, can be connected to the ppp device abstraction, using `pico_ppp_set_serial_read`, `pico_ppp_set_serial_write`, `pico_ppp_set_serial_set_speed`.

Once the physical interface is attached, the access to the remote access point gateway can be configured using `pico_ppp_set_apn`, `pico_ppp_set_username` and `pico_ppp_set_password`.

When the interface is configured, the connection may be established using `pico_ppp_connect`. Even if the peer disconnects, the connection will be brought up again automatically afterwords.

To interrupt the connection and stop the automatic reconnection, `pico_ppp_disconnect` can be called.

3.15.1 pico_ppp_create

Description

This function will create a new device association to be used with the ppp driver. The driver must then afterwards be associated with lower-level serial functions in order to be used.

Function prototype

```
struct pico_device *pico_ppp_create(void);
```

Return value

A new pico_device is allocated and returned if the device is successfully created.

Example

```
ppp = pico_ppp_create();
```

3.15.2 pico_ppp_set_serial_read

Description

This function will associate the read function from an external source (e.g. a UART device API) to the read functionality of the PPP driver. Setting up a proper read/write interface is necessary for the PPP driver to work properly.

The function associated with the read must be non-blocking, no matter the execution model of the system.

Function prototype

```
int pico_ppp_set_serial_read(struct pico_device *dev, int (*sread)(struct pico_device
*, void *, int))
```

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.
- `sread` - a pointer to a function of type `int fn(struct pico_device *, void *, int)` specifying the target serial read function. The function prototype will be called with the

device pointer, a buffer to be filled with serial data, and the maximum lenght of the usable buffer.

Return value

0 returned if the serial read function is successfully associated.

Example

```
static int my_serial_read(struct pico_device *dev, void *buf, int len)
{
    return nonblock_uart_read(buf, len);
}

pico_ppp_set_serial_read(ppp, my_serial_read);
```

3.15.3 pico_ppp_set_serial_write

Description

This function will associate the write function from an external source (e.g. a UART device API) to the write functionality of the PPP driver. Setting up a proper read/write interface is necessary for the PPP driver to work properly.

The function associated with the write must be non-blocking, no matter the execution model of the system.

Function prototype

`int pico_ppp_set_serial_write(struct pico_device *dev, int (*swrite)(struct pico_device *, const void *, int))`

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.
- `swrite` - a pointer to a function of type `int fn(struct pico_device *, const void *, int)` specifying the target serial write function. The function prototype will be called with the device pointer, a buffer to be filled with serial data, and the maximum lenght of the usable buffer.

Return value

0 returned if the serial write function is successfully associated.

Example

```
static int my_serial_write(struct pico_device *dev, const void *buf, int len)
{
    return nonblock_uart_write(buf, len);
}

pico_ppp_set_serial_write(ppp, my_serial_write);
```

3.15.4 pico_ppp_set_serial_set_speed

Description

This function will associate the set_speed function from an external source (e.g. a UART device API) to dynamically set the UART speed for the interface with the PPP driver.

Calling this function is not mandatory for the PPP UART interface to work.

Function prototype

```
int pico_ppp_set_serial_set_speed(struct pico_device *dev, int (*sset_speed)(struct pico_device *, uint32_t))
```

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.
- `sset_speed` - a pointer to a function of type `int fn(struct pico_device *, uint32_t speed)` specifying the target serial set_speed function. The function prototype will be called with the device pointer and the speed at which the UART should be configured by PPP.

Return value

0 returned if the serial set_speed function is successfully associated.

Example

```
static int my_serial_set_speed(struct pico_device *dev, uint32_t speed)
{
    return uart_set_speed(speed);
}

pico_ppp_set_serial_set_speed(ppp, my_serial_set_speed);
```

3.15.5 pico_ppp_set_apn

Description

This function allows the configuration of the APN name in order for PPP to correctly establish the connection to the remote Access Point gateway.

Function prototype

```
int pico_ppp_set_apn(struct pico_device *dev, const char *apn);
```

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.
- `apn` - a string containing the Access Point Name.

Return value

0 returned if the APN is correctly configured.

Example

```
ret = pico_ppp_set_apn(dev, "internet.apn.name");
```

3.15.6 pico_ppp_set_username

Description

This function will set an username for the PAP/CHAP authentication mechanism.

Function prototype

```
int pico_ppp_set_username(struct pico_device *dev, const char *username);
```

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.
- `username` - a string specifying the desired username.

Return value

0 returned if the username is successfully configured.

Example

```
ret = pico_ppp_set_username(dev, "john");
```

3.15.7 pico_ppp_set_password

Description

This function will set the password for the PAP/CHAP authentication mechanism.

Function prototype

```
int pico_ppp_set_password(struct pico_device *dev, const char *password);
```

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.
- `username` - a string specifying the desired password.

Return value

0 returned if the password is successfully configured.

Example

```
ret = pico_ppp_set_password(dev, "secret");
```

3.15.8 pico_ppp_connect

Description

This function will enable the PPP connection, by triggering the startup of the handshakes required at all levels. If the connection is dropped, the system will try to reconnect by restarting the handshakes, until `pico_ppp_disconnect` is finally called.

Function prototype

```
int pico_ppp_connect(struct pico_device *ppp)
```

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.

Return value

0 returned if the device is successfully connecting.

Example

```
ret = pico_ppp_connect(dev);
```

3.15.9 pico_ppp_disconnect

Description

This function will disable the PPP connection, by triggering a disconnection, and by disabling the reconnect feature, if enabled.

Function prototype

```
int pico_ppp_disconnect(struct pico_device *ppp)
```

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.

Return value

0 returned if the device is successfully put in disconnected state.

Example

```
ret = pico_ppp_disconnect(dev);
```

3.16 Optimized Link State Routing (OLSR) Module

OLSR is a proactive routing protocol for mobile ad-hoc networks (MANETs). It is well suited to large and dense mobile networks, as the optimization achieved using the MPRs works well in this context. The larger and more dense a network, the more optimization can be achieved as compared to the classic link state algorithm. OLSR uses hop-by-hop routing, i.e., each node uses its local information to route packets.

OLSR is well suited for networks, where the traffic is random and sporadic between a larger set of nodes rather than being almost exclusively between a small specific set of nodes. As a proactive protocol, OLSR is also suitable for scenarios where the communicating pairs change over time: no additional control traffic is generated in this situation since routes are maintained for all known destinations at all times. – cfr. RFC3626

3.16.1 pico_olsr_add

Description

This function will add the target device to the OLSR mechanism on the machine, meaning that it will be possible to advertise and collect routing information using Optimized Link State Routing protocol, as described in RFC3626, through the target device.

In order to use multiple devices in the OLSR system, this function needs to be called multiple times, once per device.

Function prototype

```
pico_olsr_add(struct pico_device *dev);
```

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.

Return value

0 returned if the device is successfully added.

Example

```
ret = pico_olsr_add(dev);
```

3.17 Ad-hoc On-Demand Distance Vector Routing (AODV)

AODV is a reactive routing protocol for mobile ad-hoc networks (MANETs). Its best fit are especially ultra-low power radio networks, or those RF topologies where sporadic traffic between a small specific set of nodes is foreseen. In order to create a route, one node must explicitly start the communication towards a remote node, and the route is created ad-hoc upon the demand for a specific network path. AODV guarantees that the traffic generated by each node in order to create and maintain routes is kept as low as possible.

3.17.1 pico_aodv_add

Description

This function will add the target device to the AODV mechanism on the machine, meaning that it will be possible to advertise and collect routing information using Ad-hoc On-Demand Distance Vector Routing, as described in RFC3561, through the target device.

In order to use multiple devices in the AODV system, this function needs to be called multiple times, once per device.

Function prototype

```
pico_aodv_add(struct pico_device *dev);
```

Parameters

- `dev` - a pointer to a struct `pico_device` specifying the target interface.

Return value

0 returned if the device is successfully added.

Example

```
ret = pico_aodv_add(dev);
```

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, read, &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 += read;
    } 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

```
/* identifier struct */
struct dns_identifier {
  uint8_t id;
  /* ... */
};

/* callback function of URL translation */
void cb_getaddr(char *ip, void *arg)
{
  struct dns_identifier *id_getaddr = (struct dns_identifier *) arg;

  /* NULL indicates an error condition */
  if (!ip) {
    printf("DNS error occured: %s\n", strerror(pico_err));
    return;
  }
  printf("DNS translation to ip %s (id %u)\n", ip, id_getaddr ? id_getaddr->id : 0);

  /* important: free the received pointers! */
  PICO_FREE(ip);
  if (id_getaddr)
    PICO_FREE(id_getaddr);
}

/* callback function of IP translation */
void cb_getname(char *url)
{
  struct dns_identifier *id_getname = (struct dns_identifier *) arg;

  /* NULL indicates an error condition */
  if (!url) {
    printf("DNS error occured: %s\n", strerror(pico_err));
    return;
  }
  printf("DNS translation to url %s (id %u)\n", url, id_getname ? id_getname->id : 0);

  /* important: free the received pointers! */
  PICO_FREE(url);
  if (id_getname)
```

```
    PICO_FREE(id_getname);
}

/* initialize the dns */
void app_dns(char *url, char *ip)
{
  struct pico_ip4 nameserver = { };
  struct dns_identifier *id_getaddr = NULL, *id_getname = NULL;

  /* 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 */
  id_getaddr = PICO_ZALLOC(sizeof(struct dns_identifier));
  id_getaddr->id = 1;
  pico_dns_client_getaddr(url, &cb_getaddr, id_getaddr);

  /* request translation of IP f.e. 8.8.8.8 */
  id_getname = PICO_ZALLOC(sizeof(struct dns_identifier));
  id_getname->id = 2;
  pico_dns_client_getname(ip, &cb_getname, id_getname);
}
```

4.6 DHCP client example

```
int main(void)
{
  uint8_t mac_eth0[6] = {0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc};
  uint8_t mac_eth1[6] = {0xcb, 0xa9, 0x87, 0x65, 0x43, 0x21};
  char s_addr_eth0[16] = { }, s_addr_eth1[16] = { };
  void *identifier_eth0 = NULL, *identifier_eth1 = NULL;
  uint32_t xid_eth0 = 0, xid_eth1 = 0;
  struct pico_device *eth0 = NULL, *eth1 = NULL;
  struct pico_ip4 addr_eth0 = { }, addr_eth1 = { };

  /* see section 2.5 Network devices integration */
  eth0 = pico_device_create("eth0", mac_eth0);
  eth1 = pico_device_create("eth1", mac_eth1);

  pico_stack_init();

  if (pico_dhcp_initiate_negotiation(eth0, &cb_dhcpclient, &xid_eth0) < 0) {
      printf("DHCPC: error initiating negotiation: %s\n", strerror(pico_err));
      exit(255);
  }
  if (pico_dhcp_initiate_negotiation(eth1, &cb_dhcpclient, &xid_eth1) < 0) {
      printf("DHCPC: error initiating negotiation: %s\n", strerror(pico_err));
```

```
      exit(255);
  }

  for(;;) {
    pico_stack_tick();
    /* did both devices get a successful lease? */
    if (xid_eth0 && xid_eth1)
      break;
    PICO_IDLE();
  }

  identifier_eth0 = pico_dhcp_get_identifier(xid_eth0);
  addr_eth0 = pico_dhcp_get_address(identifier_eth0);
  pico_ipv4_to_string(s_addr_eth0, addr_eth0.addr);
  printf("Device %s got leased IP %s\n", eth0->name, s_addr_eth0);

  identifier_eth1 = pico_dhcp_get_identifier(xid_eth1);
  addr_eth1 = pico_dhcp_get_address(identifier_eth1);
  pico_ipv4_to_string(s_addr_eth1, addr_eth1.addr);
  printf("Device %s got leased IP %s\n", eth1->name, s_addr_eth1);

  return 0;
}
```

4.7 HTTP Client example

```
static char *url_filename = NULL;

static int http_save_file(void *data, int len)
{
  int fd = open(url_filename, O_WRONLY |O_CREAT | O_TRUNC, 0660);
  int w, e;
  if (fd < 0)
    return fd;

  printf("Saving data to : %s\n",url_filename);
  w = write(fd, data, len);
  e = errno;
  close(fd);
  errno = e;
  return w;
}
void wget_callback(uint16_t ev, uint16_t conn)
{
  char data[1024 * 1024]; // MAX: 1M
  static int _length = 0;

  if(ev & EV_HTTP_CON)
```

```
{
  printf(">>> Connected to the client \n");
  /* you can let the client use the default generated header
     or you can create you own string header (compatible with HTTP/1.x */
  pico_http_client_sendHeader(conn,NULL,HTTP_HEADER_DEFAULT);
}

if(ev & EV_HTTP_REQ)
{
  struct pico_http_header * header = pico_http_client_readHeader(conn);
  printf("Received header from server...\n");
  printf("Server response : %d\n",header->responseCode);
  printf("Location : %s\n",header->location);
  printf("Transfer-Encoding : %d\n",header->transferCoding);
  printf("Size/Chunk : %d\n",header->contentLengthOrChunk);
}

if(ev & EV_HTTP_BODY)
{
  int len;

  printf("Reading data...\n");
  /*
    Data is passed to you without you worrying if the transfer is
    chunked or the content-length was specified.
  */
  while((len = pico_http_client_readData(conn,data + _length,1024)))
  {
    _length += len;
  }
}

if(ev & EV_HTTP_CLOSE)
{
  struct pico_http_header * header = pico_http_client_readHeader(conn);
  int len;
  printf("Connection was closed...\n");
  printf("Reading remaining data, if any ...\n");
  while((len = pico_http_client_readData(conn,data,1000u)) && len > 0)
  {
    _length += len;
  }
  printf("Read a total data of : %d bytes \n",_length);

  if(header->transferCoding == HTTP_TRANSFER_CHUNKED)
  {
    if(header->contentLengthOrChunk)
    {
      printf("Last chunk data not fully read !\n");
```

```
        exit(1);
      }
      else
      {
        printf("Transfer ended with a zero chunk! OK !\n");
      }
    } else
    {
      if(header->contentLengthOrChunk == _length)
      {
        printf("Received the full : %d \n",_length);
      }
      else
      {
        printf("Received %d , waiting for %d\n",_length, header->contentLengthOrChunk);
        exit(1);
      }
    }

    if (!url_filename) {
      printf("Failed to get local filename\n");
      exit(1);
    }

    if (http_save_file(data, _length) < _length) {
      printf("Failed to save file: %s\n", strerror(errno));
      exit(1);
    }
    pico_http_client_close(conn);
    exit(0);
  }

  if(ev & EV_HTTP_ERROR)
  {
    printf("Connection error (probably dns failed : check the routing table), trying to clos
    pico_http_client_close(conn);
    exit(1u);
  }

  if(ev & EV_HTTP_DNS)
  {
    printf("The DNS query was successful ... \n");
  }
}

void app_wget(char *arg)
{
  char * url;
  cpy_arg(&url, arg);
```

```
  if(!url)
  {
    fprintf(stderr, " wget expects the url to be received\n");
    exit(1);
  }

  // when opening the http client it will internally parse the url passed
  if(pico_http_client_open(url,wget_callback) < 0)
  {
    fprintf(stderr," error opening the url : %s, please check the format\n",url);
    exit(1);
  }
  url_filename = basename(url);
}
```

4.8 HTTP Server example

```
#define SIZE 4*1024

void serverWakeup(uint16_t ev,uint16_t conn)
{
  static FILE * f;
  char buffer[SIZE];

  if(ev & EV_HTTP_CON)
  {
      printf("New connection received....\n");
      pico_http_server_accept();
  }

  if(ev & EV_HTTP_REQ) // new header received
  {
    int read;
    char * resource;
    printf("Header request was received...\n");
    printf("> Resource : %s\n",pico_http_getResource(conn));
    resource = pico_http_getResource(conn);

    if(strcmp(resource,"/")==0 || strcmp(resource,"index.html") == 0 || strcmp(resource,"/in
    {
          // Accepting request
          printf("Accepted connection...\n");
          pico_http_respond(conn,HTTP_RESOURCE_FOUND);
          f = fopen("test/examples/index.html","r");

          if(!f)
```

```
        {
          fprintf(stderr,"Unable to open the file /test/examples/index.html\n");
          exit(1);
        }

        read = fread(buffer,1,SIZE,f);
        pico_http_submitData(conn,buffer,read);
  }
  else
  { // reject
    printf("Rejected connection...\n");
    pico_http_respond(conn,HTTP_RESOURCE_NOT_FOUND);
  }

}

if(ev & EV_HTTP_PROGRESS) // submitted data was sent
{
  uint16_t sent, total;
  pico_http_getProgress(conn,&sent,&total);
  printf("Chunk statistics : %d/%d sent\n",sent,total);
}

if(ev & EV_HTTP_SENT) // submitted data was fully sent
{
  int read;
  read = fread(buffer,1,SIZE,f);
  printf("Chunk was sent...\n");
  if(read > 0)
  {
      printf("Sending another chunk...\n");
      pico_http_submitData(conn,buffer,read);
  }
  else
  {
      printf("Last chunk !\n");
      pico_http_submitData(conn,NULL,0);// send the final chunk
      fclose(f);
  }
}

if(ev & EV_HTTP_CLOSE)
{
  printf("Close request...\n");
  pico_http_close(conn);
}

if(ev & EV_HTTP_ERROR)
{
```

```
    printf("Error on server...\n");
    pico_http_close(conn);
  }
}
/* simple server example that serves the index(.html) page */
void app_httpd(char *arg)
{
  /* transfer encoding with this server is always chunked and you can
     submit chunks to the client, without needing to specify the content-length of the
     body response */
  if( pico_http_server_start(0,serverWakeup) < 0)
  {
    fprintf(stderr,"Unable to start the server on port 80\n");
  }
}
```

4.9 TFTP Client (application driven)

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <inttypes.h>
#include "pico_stack.h"
#include "pico_config.h"
#include "pico_ipv4.h"
#include "pico_icmp4.h"
#include "pico_socket.h"
#include "pico_stack.h"
#include "pico_device.h"
#include "pico_dev_vde.h"
#include "pico_tftp.h"

static struct pico_device *pico_dev;

int32_t get_filesize(const char *filename)
{
    int ret;
    struct stat buf;

    ret = stat(filename, &buf);
    if (ret)
        return -1;
    return buf.st_size;
}

void start_rx(struct pico_tftp_session *session, int *synchro,
              const char *filename, int options)
```

```
{
    int ret;
    int fd;
    int32_t len;
    uint8_t buf[PICO_TFTP_PAYLOAD_SIZE];
    int left = 1000;
    int countdown = 0;

    printf("Start receiving file %s with options set to %d\n", filename, options);

    if (options) {
        ret = pico_tftp_set_option(session, PICO_TFTP_OPTION_FILE, 0);
        if (ret) {
            fprintf(stderr, "Error in pico_tftp_set_option\n");
            exit(1);
        }
    }

    ret = pico_tftp_app_start_rx(session, filename);
    if (ret) {
        fprintf(stderr, "Error in pico_tftp_app_start_rx\n");
        exit(1);
    }

    fd = open(filename, O_WRONLY | O_EXCL | O_CREAT, 0664);
    if (!fd) {
        fprintf(stderr, "Error in open\n");
        countdown = 1;
    }

    for(;left; left -= countdown) {
        usleep(2000); //PICO_IDLE();
        pico_stack_tick();
        if (countdown)
            continue;

        if (*synchro) {
            len = pico_tftp_get(session, buf, PICO_TFTP_PAYLOAD_SIZE);
            if (len < 0) {
                fprintf(stderr, "Failure in pico_tftp_get\n");
                close(fd);
                countdown = 1;
                continue;
            }
            ret = write(fd, buf, len);
            if (ret < 0) {
                fprintf(stderr, "Error in write\n");
                pico_tftp_abort(session, TFTP_ERR_EXCEEDED, "File write error");
                close(fd);
```

```
                countdown = 1;
                continue;
            }
            printf("Written %" PRId32 " bytes to file (synchro=%d)\n", len, *synchro);

            if (len != PICO_TFTP_PAYLOAD_SIZE) {
                close(fd);
                printf("Transfer complete!\n");
                countdown = 1;
            }
        }
    }
}

void start_tx(struct pico_tftp_session *session, int *synchro,
              const char *filename, int options)
{
    int ret;
    int fd;
    int32_t len;
    uint8_t buf[PICO_TFTP_PAYLOAD_SIZE];
    int left = 1000;
    int countdown = 0;

    printf("Start sending file %s with options set to %d\n", filename, options);

    if (options) {
        ret = get_filesize(filename);
        if (ret < 0) {
            fprintf(stderr, "Error in get_filesize\n");
            exit(1);
        }

        ret = pico_tftp_set_option(session, PICO_TFTP_OPTION_FILE, ret);
        if (ret) {
            fprintf(stderr, "Error in pico_tftp_set_option\n");
            exit(1);
        }
    }

    ret = pico_tftp_app_start_tx(session, filename);
    if (ret) {
        fprintf(stderr, "Error in pico_tftp_app_start_rx\n");
        exit(1);
    }

    fd = open(filename, O_RDONLY, 0444);
    if (!fd) {
        fprintf(stderr, "Error in open\n");
```

```
        pico_tftp_abort(session, TFTP_ERR_EACC, "Error opening file");
        countdown = 1;
    }

    for(;left; left -= countdown) {
        usleep(2000); //PICO_IDLE();
        pico_stack_tick();
        if (countdown)
            continue;

        if (*synchro) {
            ret = read(fd, buf, PICO_TFTP_PAYLOAD_SIZE);
            if (ret < 0) {
                fprintf(stderr, "Error in read\n");
                pico_tftp_abort(session, TFTP_ERR_EACC, "File read error");
                close(fd);
                countdown = 1;
                continue;
            }
            printf("Read %" PRId32 " bytes from file (synchro=%d)\n", len, *synchro);

            len = pico_tftp_put(session, buf, ret);
            if (len < 0) {
                fprintf(stderr, "Failure in pico_tftp_put\n");
                close(fd);
                countdown = 1;
                continue;
            }

            if (len != PICO_TFTP_PAYLOAD_SIZE) {
                close(fd);
                printf("Transfer complete!\n");
                countdown = 1;
            }
        }
    }
}

void usage(const char *text)
{
    fprintf(stderr, "%s\nArguments must be <filename> <mode>\n"
                        "<mode> can be:\n"
                        "\tg => GET request without options\n"
                        "\tG => GET request WITH options\n"
                        "\tp => PUT request without options\n"
                        "\tP => PUT request WITH options\n\n",
            text);
    exit(1);
}
```

```
int main(int argc, char** argv)
{
    struct pico_ip4 my_ip;
    union pico_address server_address;
    struct pico_ip4 netmask;
    struct pico_tftp_session *session;
    int synchro;
    int options = 0;
    void (*operation)(struct pico_tftp_session *session, int *synchro,
                      const char *filename, int options);

    unsigned char macaddr[6] = {
        0, 0, 0, 0xa, 0xb, 0x0
    };

    uint16_t *macaddr_low = (uint16_t *) (macaddr + 2);
    *macaddr_low = *macaddr_low ^ (uint16_t)((uint16_t)getpid() & (uint16_t)0xFFFFU);
    macaddr[4] ^= (uint8_t)(getpid() >> 8);
    macaddr[5] ^= (uint8_t) (getpid() & 0xFF);

    pico_string_to_ipv4("10.40.0.10", &my_ip.addr);
    pico_string_to_ipv4("255.255.255.0", &netmask.addr);
    pico_string_to_ipv4("10.40.0.2", &server_address.ip4.addr);

    if (argc != 3) {
        usage("Invalid number or arguments");
    }

    switch (argv[2][0]) {
    case 'G':
        options = 1;
    case 'g':
        operation = start_rx;
        break;
    case 'P':
        options = 1;
    case 'p':
        operation = start_tx;
        break;
    default:
        usage("Invalid mode");
    }

    printf("%s start!\n", argv[0]);
    pico_stack_init();
    pico_dev = (struct pico_device *) pico_vde_create("/tmp/vde_switch",
                                                      "tap0", macaddr);
```

```
    if(!pico_dev) {
        fprintf(stderr, "Error creating pico device,"
                        " got enough privileges? Exiting...\n");
        exit(1);
    }

    pico_ipv4_link_add(pico_dev, my_ip, netmask);
    printf("Starting picoTCP loop\n");

    session = pico_tftp_app_setup(&server_address, short_be(PICO_TFTP_PORT),
                                  PICO_PROTO_IPV4, &synchro);
    if (!session) {
        fprintf(stderr, "Error in pico_tftp_app_setup\n");
        exit(1);
    }

    printf("synchro %d\n", synchro);

    operation(session, &synchro, argv[1], options);
}
```

A. Supported RFC's

RFC	Description
RFC 768	User Datagram Protocol (UDP)
RFC 791	Internet Protocol (IP)
RFC 792	Internet Control Message Protocol (ICMP)
RFC 793	Transmission Control Protocol (TCP)
RFC 816	Fault Isolation and Recovery
RFC 826	Address Resolution Protocol (ARP)
RFC 879	The TCP Maximum Segment Size and Related Topics
RFC 894	IP over Ethernet
RFC 896	Congestion Control in IP/TCP Internetworks
RFC 919	Broadcasting Internet Datagrams
RFC 922	Broadcasting Internet Datagrams in the Presence of Subnets
RFC 950	Internet Standard Subnetting Procedure
RFC 1009	Requirements for Internet Gateways
RFC 1034	Domain NamesConcepts and Facilities
RFC 1035	Domain NamesImplementation and Specification
RFC 1071	Computing the Internet Checksum
RFC 1112	Internet Group Management Protocol (IGMP)
RFC 1122	Requirements for Internet HostsCommunication Layers
RFC 1123	Requirements for Internet Hosts - Application and Support ([1])
RFC 1191	Path MTU Discovery ([1])
RFC 1323	TCP Extensions for High Performance
RFC 1332	The PPP Internet Protocol Control Protocol (IPCP)
RFC 1334	PPP Authentication Protocols
RFC 1337	TIME-WAIT Assassination Hazards in TCP
RFC 1350	The TFTP Protocol (Revision 2)
RFC 1534	Interoperation Between DHCP and BOOTP
RFC 1542	Clarifications and Extensions for the Bootstrap Protocol
RFC 1661	The Point-to-Point Protocol (PPP)
RFC 1662	PPP in HDLC-like Framing
RFC 1812	Requirements for IP Version 4 Routers
RFC 1878	Variable Length Subnet Table For IPv4
RFC 1886	DNS Extensions to Support IP Version 6 ([1])
RFC 1994	PPP Challenge Handshake Authentication Protocol (CHAP)
RFC 2018	TCP Selective Acknowledgment Options
RFC 2131	Dynamic Host Configuration Protocol (DHCP)
RFC 2132	DHCP Options and BOOTP Vendor Extensions
RFC 2236	Internet Group Management Protocol, Version 2
RFC2347	TFTP Option Extension
RFC 2349	TFTP Timeout Interval and Transfer Size Options
RFC 2460	Internet Protocol, Version 6 (IPv6) Specification
RFC 2581	TCP Congestion Control

RFC 2663	IP Network Address Translator (NAT) Terminology and Considerations
RFC 2710	Multicast Listener Discovery (MLD) for IPv6
RFC 3042	Enhancing TCP's Loss Recovery Using Limited Transmit
RFC 3315	Dynamic Host Configuration Protocol for IPv6 (DHCPv6) ([1])
RFC 3376	Internet Group Management Protocol, Version 3
RFC 3517	A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP
RFC 3561	Ad-hoc On-Demand Distance Vector (AODV) Routing
RFC 3626	Optimized Link State Routing Protocol (OLSR)
RFC 3782	The NewReno Modification to TCP's Fast Recovery Algorithm
RFC 3810	Multicast Listener Discovery Version 2 (MLDv2) for IPv6
RFC 3927	Dynamic Configuration of IPv4 Link-Local Addresses
RFC 4291	IP Version 6 Addressing Architecture
RFC 4443	Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification
RFC 4861	Neighbor Discovery for IP version 6 (IPv6)
RFC 4862	IPv6 Stateless Address Autoconfiguration
RFC 6691	TCP Options and Maximum Segment Size (MSS)
RFC 6762	Multicast DNS
RFC 6763	DNS-based Service Discovery

([1]) Work in progress ([2]) Experimental