PicOS Virtual Transceiver Interface

Version 1.0
September 22, 2003

Preamble

This document outlines the implementation of the plugin-extensible, open-ended, generic transceiver interface for PicOS. The purpose of this interface, dubbed TCV, is to provide a simple collection of API, independent of the underlying implementation of networking, which, in addition to enabling a rapid deployment of networked application for eCOG-driven cards would make it easy to develop testbeds using emulated radio interfaces. To avoid the protocol layering problems haunting small footprint solutions, the presented interface is essentially layer-less and its semi-complete generic functionality can be redefined by plugins. Also, the actual implementation of the physical interface to the network can be encapsulated into a relatively simple and easily exchangeable module. To facilitate development, testing, and experiments, multiple plugins and physical interfaces can coexist within the same system configuration.

PicOS is a sister document, referred to as [PicOS].

1. TCV structure

The structure of TCV and its relationship with other system components is shown in Figure 1. In essence, the module implements transparent management of buffer (packet) storage organized into a dynamic number of queues, timeouts definable on a per-packet basis, multiple application access points (roughly equivalent to connections or sessions), and provides a unified set of functions for interfacing plugins and physical modules.

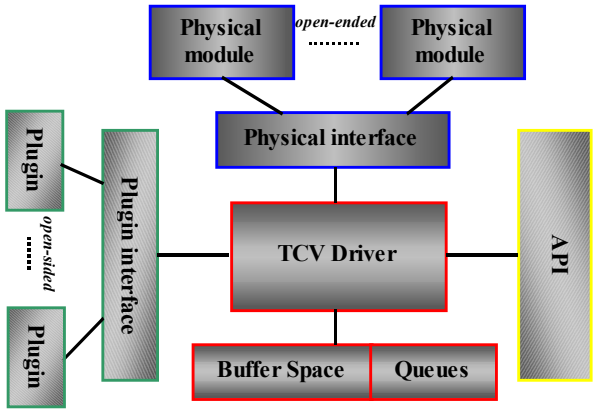

Figure 1. The structure of TCV.

The API of TCV can be viewed as an abstract device, which in principle could be made accessible via the standard `io` mechanism of PicOS. Due to some idiosyncrasies of this *device*, we decided to use a special set of TCV-specific functions, instead of overloading the `io` operation with numerous CONTROL requests. This detail is purely technical and provided as a shortcut. In fact, TCV (with the assistance of plugins) is capable of creating equivalents of multiple devices, e.g., representing simultaneous, possibly independent, network connections.

A workable TCV setup involves at least one physical module and at least one plugin. The application can open a connection, receive and transmit packets over a connection, close a connection, without having to worry about details, like allocation/deallocation of buffer space, header/trailer processing, and possibly error recovery, which operations are performed transparently by the plugin(s). A single plugin can cooperate with multiple physical modules, multiple plugins can claim frames delivered by different physical modules for their specific processing, and so on. In particular, it should be possible to implement the TCP/IP stack as a plugin (or possibly a collection of plugins). As, regardless of the actual configuration of plugins and physical modules configured into the system, the API provided by TCV is essentially fixed, we can say that TCV, in addition to being open-ended (on the physical end) is open-sided (on the plugin side), while being closed with respect to the API (i.e., on the application's end).

2. Physical interface

Physical modules are interfaced with TCV via a collection of five functions whose headers are defined in `tcvphys.h`. Three sample physical modules provided in the library (file names starting with "`phys_`") can be used as illustrations how to build such modules. During initialization, a physical module registers itself with TCV by executing this function:

```
word tcvphy_reg (int id, ctrlfun_t cfun, int info)
```

whose first argument is a small integer number uniquely identifying the module and the second argument provides a control function for affecting the specific parameters of the module. The module identifier allows a plugin to recognize a specific physical module, e.g., as the source of an inbound packet. Also, as a parameter of the open operation available to the application, allows it to associate a specific session with a specific physical module (while another parameter identifies a specific plugin). Module numbers are usually very small. They are indexes into an array of physical modules, whose maximum size is determined by a configuration parameter (`TCV_MAX_PHYS` in `options.h` with the default value of 3 and the maximum of 8).

The third argument of `tcvphy_reg` provides the interface *information* attribute i.e., an integer value that is supposed to be unique among all possible interfaces, and whose role is to identify the module uniquely (regardless of its short `id`) for the plugin or application.

The control function should have the following header:

```
int cntrlfun (int option, address value)
```

with the first argument identifying the option of the physical module, and the second providing its (new) value (which can be a structure pointed to by the specified address). Some options can be considered standard (e.g., switching on and off the receiver/transmitter), while some others may be strongly interface-specific (e.g., switching on the promiscuous mode of the Ethernet chip). Some calls to the control function may be intended to read the value of an option rather than changing it. This value can be returned via the function value or via the parameter (viewed as a pointer). The TCV API gives the application access to the control functions of the physical modules handling its connections. If the application decides to take advantage of an exotic option, it should know what it is doing.

A registered physical module is assigned a single queue of outgoing packets, which is handled automatically by TCV. The value returned by `tcvphy_reg` is a 16-bit word identifying the *queue event* that will be triggered by TCV whenever the outgoing queue becomes nonempty. The physical module can acquire the top packet from its outgoing queue via this function:

```
address tcvphy_get (int id, int *len)
```

The first argument is the module identifier that was specified with `tcvphy_reg`. If there is a packet queued for transmission by the physical module, the function returns a pointer to the first packet in the queue (the packet string), while the length of this string is returned via the second argument. Note that packet strings (as perceived by physical modules) are always word-aligned, although their length is expressed in bytes (and it doesn't have to be even). The function returns `NULL`, if there is no packet to transmit at this time. Typically, `tcvphy_get` is called from a process of the physical module, e.g., after responding to the queue event.

The queue event is triggered when the outgoing queue of packets becomes nonempty, and also when an *urgent* packet (see section 3) is put into the (possibly nonempty) queue, necessarily at the front.

It is possible to check if the outgoing queue of the physical module is not empty by calling this function:

```
int tcvphy_top (int id)
```

with the module identifier as the argument. The function returns 0 if the queue is empty, 1 if the queue contains at least one packet, or 2 if the queue contains at least one urgent packet. In the latter case, such a packet necessarily appears at the front of the queue, so it will be acquired with a subsequent call to `tcvphy_get`.

The packet whose pointer is returned by `tcvphy_get` is stored in the dynamic buffer pool handled by TCV. This means that the physical module must tell TCV when it is done with the packet (note that a physical packet transmission may involve blocking and

take a nontrivial amount of time), such that its buffer can be deallocated. This is accomplished by the following function:

```
tcvphy_end (address packet)
```

which accepts the original packet handle (pointer) that was produced by `tcvphy_get` and notifies TCV that the packet has been transmitted.

Determining the fate of the packet that has been transmitted by the physical module is up to the plugin, and a call to `tcvphy_end` does not imply that the packet has been erased and forgotten. For example, the plugin may decide to keep the packet for a possible retransmission until it is acknowledged by the recipient.

The reception end is handled in a very simple manner. When the physical module receives a packet that should be passed to TCV, it calls the following function:

```
tcvphy_rcv (int id, address buf, int len)
```

whose last two arguments describe the location of the packet. Note that the physical module is responsible for allocating and handling its own reception buffer. The above operation never blocks and it expects that TCV is able to copy the packet to its internal buffer storage, furnishing it with internal attributes based on the decision of the plugin. Thus, following the call to `tcvphy_rcv`, the physical module is free to reuse the reception buffer for whatever it pleases.

```
int tcvphy_erase (int id)
```

This operation drains the output queue of the interface module and returns the number of erased packets. It can be viewed as a combination of `tcvphy_get` and `tcvphy_end` executed in a loop until the queue becomes empty. The erased packets are formally treated as if they have been transmitted, i.e., the `tcv_xmt` function of the plugin is called for every erased packet (see section 4).

3. Application interface

Before the application can use the services offered by TCV (in collaboration with its registered plugins and physical modules), it must open a session. The exact semantics of what it means to *"open a session"* depends on the plugin. In the simplest case, as implemented in the sample *null* plugin (file `plug_null.c`), there can be a single session per physical interface representing a more or less direct connection to the network, but nonetheless, this session must be explicitly set up, i.e., opened. This is accomplished by the following operation:

```
int tcv_open (word state, int phid, int plid, ...)
```

whose exact configuration of arguments depends on the capabilities of the plugin responsible for handling the session. The first argument identifies a state in the calling process. Depending on how the session is organized (e.g., the operation may involve

setting up a TCP connection to some remote host), the open request may block. In such a case, the first argument indicates the state in which the process will be restarted when it makes sense to resume the operation. Similar to a standard `io` request, the state argument can be `NONE`, in which case the operation never blocks, but returns its status. If the value returned by `tcv_open` is -2 (`BLOCKED`), it means that the open operation has started but the request has not been completed. The function must be called again at a later time. The state of a pending open request is maintained by TCV (and the plugin), so it is legal to re-execute a previously blocked `tcv_open` at any time to check whether it has completed. When `state` is not `NONE`, the process is automatically blocked and restarted at the indicated state when the session status is clear.

The remaining two fixed arguments of `tcv_open` identify the physical module and the plugin responsible for handling the session. Similar to the physical identifier, the plugin identifier is a small integer number uniquely assigned to a single plugin.

Upon a successful completion, `tcv_open` returns a small integer value (starting from zero), which identifies the session and can be viewed as a file descriptor (in UNIX terminology). If the value returned by the function is -1 (`ERROR`), it means that the attempt to set up the session has failed. The reason for this failure is not formal (in which case the program would simply crash on a system error), but may have resulted from a lack of resources (too many sessions) or a higher-level problem encountered by the plugin.

The maximum number of sessions that can be opened simultaneously at any moment is determined by `TCV_MAX_DESC` with the default value of 16. The maximum setting of this symbol, which can be redefined in `options.h`, is 128. Each session is assigned an incoming queue of packets from which the application can receive data. Outgoing packets are passed to TCV without being explicitly queued within the session context.

Once opened, a session can be closed with the following function:

```
int tcv_close (word state, int fd)
```

whose second argument is a session descriptor[1] that was returned by `tcv_open`. The operation can legitimately block, in which case the first argument indicates the state to be assumed when it makes sense to reissue the request. As in the case of `tcv_open`, the half closed state of a connection is stored by TCV. In those circumstances when the operation can possibly block, such a half-closed state will remain pending (rendering the session descriptor wasted) until the application executes `tcv_close` successfully.

The function returns 0 upon success, -2 (`BLOCKED`) when the close request is pending (this is only possible if `state` is `NONE`), and -1 (`ERROR`) when there has been a high-level error detected by the plugin.

The operations of reading from and writing to the session descriptor deal with packets, while making it possible to extract/send information in smaller chunks. This simple way,

[1] In all subsequent occurrences, `fd` will stand for a session descriptor.

the structure of packets is made visible enough to account for those circumstances when the application prefers to see them explicitly, and transparent enough to view a session descriptor as a stream-like device handling (almost) unpackaged sequences of bytes.

The following operation acquires the next packet queued for reception at the session:

```
address tcv_rnp (word state, int fd)
```

and returns the packet handle (a word-aligned pointer to the first byte of the packet). Of course, the operation may block, if no packet is available in the session's incoming queue, in which case the `state` argument is interpreted in the usual way. If `state` is `NONE`, the function returns NULL if there's no packet to read.

The packet handle returned by `tcv_rnp`, besides providing immediate access to the packet string, additionally represents a more elaborate data structure maintained by TCV to keep track of various packet attributes. In particular, although the length of the received packet is not directly returned by `tcv_rnp`, this (and some other) information is readily accessible through the packet handle (see below).

The packet acquired by `tcv_rnp` is removed from the session's queue. This has two implications. First, it is possible to read (extract from the queue) multiple packets before actually processing them (e.g., from multiple processes). Second, the handle returned by `tcv_rnp` becomes the only *reference* to the received packet. Consequently, the application is responsible for indicating explicitly the moment when the packet has been processed and is not needed any more. This is accomplished by the following function:

```
void tcv_endp (address packet)
```

where the argument is a packet handle. If the handle has been acquired by `tcv_rnp` (i.e., it describes an incoming packet), it means that the application is completely done with the packet and it should be deallocated.

Outgoing packets are created in a similar way, i.e., by requesting a packet handle from TCV. This is done by the following function:

```
address tcv_wnp (word state, int fd, int length)
```

The operation builds a new packet `length` bytes long and returns its handle. The only possible reason for blocking is the lack of memory to accommodate the new packet. When this happens, the invoking process will be resumed in the indicated state when some memory becomes available. As usual, if `state` is `NONE`, the function never blocks but returns `NULL` on failure.

When the packet is filled out and complete, the application can use `tcv_endp` (see above) to notify TCV that it should be accepted as a new ready outgoing packet. Note that TCV is able to tell the difference between the two invocations of `tcv_endp` (i.e., for an incoming and an outgoing packet).

Packets handled by `tcv_rnp/tcv_wnp` are complete in terms of their size, i.e., they include room for the possible headers and/or trailers. While those headers and trailers need not be used by the application (the plugin may be solely responsible for interpreting them and filling them in), the application must be aware of them if it insists on interpreting directly the raw packet contents made accessible via handles. If the headers/trailers are processed by the plugin, the `length` parameter of `tcv_wnp` refers to the *logical* component of the packet (the application payload) excluding the parts that are not meant for the application. TCV will obtain the total required length of the packet frame by consulting the plugin. However, when the application stores its payload within the packet, it has to skip the header explicitly. If the above sounds confusing, just keep reading.

All this hassle can be avoided by accessing the packet contents (the payload) via some other functions provided by TCV. In particular, this one:

```
int tcv_read (address packet, char *buf, int len)
```

extracts up to `len` payload bytes from the incoming packet pointed to by the specified handle and stores them in `buf`. It uses internal pointers maintained by TCV which are updated after every extraction. The function returns the number of extracted bytes, which can be less than `len` (if there are not that many bytes left in the packet). In particular, if the function returns zero, it means that all the payload has been extracted from the packet. Note that `tcv_read` is intended for incoming packets, i.e., ones that have been acquired with `tcv_rnp`, and it should never be called for an outgoing packet.

Similarly, the payload of an outgoing packet can be filled using the following function:

```
int tcv_write (address packet, const char *buf, int len)
```

whose semantics should be obvious. The function returns the number of bytes stored within the payload section of the packet. When it returns zero or less than `len`, it means that the entire payload has been filled.

Another function in this basket is the following one, which can also be used to determine the length of a packet acquired by `tcv_rnp`:

```
int tcv_left (address packet)
```

It returns the number of payload bytes still left within the packet. For an incoming packet, this means the number of bytes left for extraction. Immediately after the packet has been acquired by `tcv_rnp`, this number is equal to the total length of the payload; then it is decremented after each call to `tcv_read`. For an outgoing packet, this number starts with the length specified for `tcv_wnp`, and then it is decremented by `tcv_write`.

TCV differentiates between normal-priority and urgent packets. A packet can be made urgent by the application, and then it will receive a preferential treatment by TCV. This is the function to mark a packet as urgent:

```
void tcv_urgent (address packet)
```

However, there is no way to identify an urgent incoming packet by the application (except, of course, by interpreting the packet contents).

TCV provides a way to invoke the control function of the underlying physical module of a session from the application. This is accomplished by calling:

```
int tcv_control (int fd, int option, address value)
```

which function is in essence a link to the control function declared by the physical module (see section 2). The primary difference is that from the application, the physical module is identifiable through the session descriptor, rather than directly (or via the module identifier).

Additionally, two special option values, `PHYSOPT_PLUGINFO` and `PHYSOPT_PHYSINFO`, are processed by the plugin, rather than being passed to the physical module's control function. With `PHYSOPT_PLUGINFO`, the `fd` argument is interpreted as a plugin `id`, and the function returns the plugin *information* attribute (see the next section). With `PHYSOPT_PHYSINFO`, `fd` is interpreted as a physical module identifier, and the function returns the *information* descriptor of the respective physical module (see section 2). In both cases, if the specified identifier does not correspond to an existing plugin/module, the function returns zero.

The last application-level function provided by TCV is the one for configuring plugins:

```
void tcv_plug (int id, tcvplug_t *plugin)
```

The first argument is the numerical plugin identifier whose purpose is to uniquely identify all configured plugins, e.g., for reference with `tcv_open`. The second argument points to a data structure containing seven function pointers. This structure and the purpose of all those functions are described in the next section.

4. Plugin interface

A plugin is described by a set of seven functions and one integer value, which are collectively provided via the following structure:

```
typedef struct {
   int (*tcv_ope) (int phid, int fd, va_list ap);
   int (*tcv_clo) (int phid, int fd);
   int (*tcv_rcv) (int phid, address buf, int len, int *ses,
      tcvadp_t *frm);
   int (*tcv_frm) (address packet, int phid, tcvadp_t *frm);
   int (*tcv_out) (address packet);
   int (*tcv_xmt) (address packet);
   int (*tcv_tmt) (address packet);
   int tcv_info;
} tcvplug_t;
```

The single numerical attribute (tcv_info) is the plugin's global *information descriptor* intended to be unique among all plugins.

The plugin functions may reference a number of operations offered by TCV to be used exclusively by plugins. In addition, the plugin functions can also call those functions of TCV which are intended for the application.

We start from listing the roles and responsibilities of the plugin functions.

→ **tcv_ope**

This function is called to handle the plugin end of the session open operation. Its first two arguments are, respectively, the physical module identifier and the session descriptor allocated by TCV to the new session. The third argument is a pointer to the list of all the extra arguments (the ones represented by dots in the header) that were passed to `tcv_open`.

The value returned by the function is interpreted in the following way: zero indicates success and means that the new session has been accepted by the plugin; -1 means error, i.e., the session has been rejected by the plugin; any other value means that the request has been accepted by the plugin, but the session hasn't been set up immediately. In the last case, TCV interprets the value returned by `tcv_ope` as the identifier of the event that will be triggered by the plugin when it makes sense to retry the operation. The process calling `tcv_open` will be blocked waiting for that event (unless the `state` argument of `tcv_open` was `NONE`).

The only reason why `tcv_open` (the application-level open function) may block is when the plugin `tcv_ope` function decides to block. By itself, TCV never blocks on opening a new session.

→ **tcv_clo**

This function is called to close the plugin end of a session being closed by `tcv_close`. The two arguments are, respectively, the physical module identifier and the session descriptor. The values returned by `tcv_clo` are interpreted as for `tcv_ope`.

→ **tcv_rcv**

This function is called by TCV when a packet is received from a physical module. The first three (input) arguments are: the physical module identifier, the packet buffer pointer (word aligned), and the total received length of the packet. The value returned by the function indicates whether the plugin has claimed the packet, and what should be done with the packet (the so called disposition code).

Having received a packet from a physical module (see operation `tcvphy_rcv`), TCV examines all the registered plugins calling their `tcv_rcv` functions). The plugins are examined in the reverse order of their numerical identifiers, and the first

plugin whose `tcv_rcv` returns nonzero is assumed to have claimed the packet. This way the plugins with low identifiers are treated as fall-back (or default) plugins for servicing the types of packets unclaimed by the higher-numbered plugins.

Having claimed a packet, `tcv_rcv` is expected to return some additional information via the last two arguments. The first of them is the descriptor of the session to which the packet should be assigned. This does not automatically imply that the packet will be queued for reception at that session. At this level, the session descriptor can be viewed as a tag assigned to the packet. The second return argument points to a simple structure consisting of two 16-bit numbers which determine the portion of the received packet that should be extracted and stored by TCV. The first of these numbers is interpreted as the offset from the beginning of the received packet, while the second is viewed as the offset from the end. In particular, if the entire received packet should be passed to TCV, both numbers should be zero. Note that any truncation at this level is not meant to identify and isolate the application level payload, but rather to eliminate those components of the physical header/trailer that the plugin considers useless and uninteresting. This can also be done by the physical module, which decides on the portion of the received packet that should be presented to TCV. A special plugin function, `tcv_frm` (see below), is provided to identify application-level payloads within packets, once they get past the reception/acceptance stage.

Note that the packet pointer passed to `tcv_rcv` is not a packet handle, i.e., it represents a raw sequence of bytes rather than a structure kept in a TCV buffer and adorned with the standard set of internal attributes. This is in contrast to the remaining four plugin functions, which accept packet handles. One should remember that operations like cloning packets, changing their attributes, assigning them to sessions, and so on, are only defined on packet handles.

The concept of a disposition code, as returned by `tcv_rcv`, is general and applicable to some other plugin functions and TCV operations available to plugins. These codes are as follows:

```
(0) TCV_DSP_PASS
```

This code means *skip* or *do nothing*. For example, it is used as the *no claim* indication by `tcv_rcv`.

```
(1) TCV_DSP_DROP
```

The packet should be dropped and ignored.

```
(2) TCV_DSP_RCV
```

The packet should be queued for reception at the session with which it is associated. If the packet is urgent, it will be queued at the front of the session's queue, otherwise, it will be queued at the end.

```
(3) TCV_DSP_RCVU
```

Make the packet urgent and queue it at the session (necessarily at the front of the queue).

(4) TCV_DSP_XMT

Queue the packet for transmission by the physical module with which the packet is associated. If the packet is urgent, it will be queued at the front of the interface's outgoing queue, otherwise, it will be queued at the end.

(5) TCV_DSP_XMTU

Make the packet urgent and queue it for transmission by the respective physical module.

Note that an outgoing packet created by the application is automatically associated with a physical module (the one that was assigned to the session when it was opened). Similarly, a received packet may be immediately associated with a session by `tcv_rcv`. If its immediate disposition is 2 or 3, the packet must be assigned to a legitimate (opened) session.

→ **tcv_frm**

This function is given a packet handle and a physical module identifier, and returns the offsets of the header and trailer within the packet that separate the framing information from the application payload. The structure pointed to by the third argument is the same as for `tcv_rcv` (see `sysio.h` for its declaration). The first word of this structure specifies the header offset from the beginning of the packet, while the second word gives the trailer offset from the end. The payload length is determined as total length - header offset - trailer offset.

The function can be legitimately called with the first argument being `NULL`. This happens when TCV attempts to allocate a new outgoing packet buffer for `tcv_wnp`. Note that in that case, to know how much memory should be requested for the new packet, TCV must query `tcv_frm` before the handle can point to anything sensible. Thus the function must be able to tell the framing parameters based exclusively on the physical module identifier.

The integer value returned by the function is ignored.

→ **tcv_out**

This function is called whenever a new outgoing packet is ready (the application executes `tcv_endp`) and its return value determines what should be done with the packet – according to the disposition codes listed above.

→ **tcv_xmt**

This function is called whenever a packet has been transmitted by a physical module. Its return value determines the fate of this packet.

→ tcv_tmt

This function is called for a packet whose timer goes off. Its return value determines the fate of the packet.

Each of the last three functions (returning disposition codes) accepts a packet handle as the argument. Let us emphasize once again that although `tcv_rcv` also returns a disposition code, it (exceptionally) deals with a raw sequence of bytes as opposed to a packet buffer accessible via a handle.

The names of the TCV functions that are callable by plugins start with "`tcvp_`". Below we list those functions in the somewhat accidental order of their subjective relevance. One should note that, if needed, a plugin may call any of the application level functions.

```
---> address tcvp_new (int length, int dsp, int fd)
```

This function creates a new packet `length` bytes long, associates it with session `fd` and sets its disposition code to `dsp`. The length argument refers to the complete length of the entire packet (including any headers and trailers). The packet is not automatically filled with any contents. The function returns the handle to the new packet or `NULL` if there is no enough memory to accommodate the new buffer.

If `dsp` is `TCV_DSP_PASS`, the packet is not assigned to any queue but left detached (e.g., to be stored by the plugin and used later). In that case, `fd` is not required to indicate a legitimate session (`NONE` is recommended in such circumstances).

Note that operations `tcv_read` and `tcv_write` are only applicable to packets that have been acquired (not necessarily by the application but possibly also by the plugin) via `tcv_rnp` or `tcv_wnp`. In particular, they should not be used for filling out or reading a packet created by `tcvp_new`, unless that packet has been directed to the session queue and extracted from it by `tcv_rnp`.

```
---> void tcvp_dispose (address packet, int dsp)
```

This operation explicitly sets the disposition code for a packet. It can be used, e.g., for changing the disposition code of a packet created internally by the plugin. Return codes of the plugin functions can be viewed as shortcuts for calling `tcvp_dispose` before their return.

Perhaps it makes sense to mention at this point that TCV does not implement any default dropping policy that would implicitly and automatically remove old (or less important) packets as to make room for the new (or more important) ones. It is up to the plugin to implement such a policy, if it is desired. Note that any packet can be explicitly discarded by setting its disposition code to `TCV_DSP_DROP`.

```
---> address tcvp_clone (address packet, int dsp)
```

This function creates a copy of the indicated packet, sets its disposition code to dsp, and returns a handle to the new packet. All the attributes of the clone are inherited from the original, except for the *urgent* attribute, which is cleared. The function returns `NULL` if there is no buffer space available to accommodate the clone.

```
---> int tcvp_length (address packet)
```

This function returns the total length of the indicated packet.

```
---> void tcvp_assign (address packet, int fd)
```

This function assigns the indicated packet to the given session. The second argument must describe an open session. Additionally and automatically, the packet is also assigned to the physical module associated with the session (which is the consistent thing to do). However, the following operation:

```
---> void tcvp_attach (address packet, int phid)
```

assigns the packet to the specified physical module while leaving the session attribute intact. This may lead to inconsistencies (if the packet's session is attached to a different physical module), but is OK, e.g., if the packet is to be transmitted and then dropped.

```
---> void tcvp_hook (address packet, address *hook)
```

This operation assigns a hook to the packet by storing the packet handle at the indicated `hook` location and noting this fact among the packet's attributes. If the packet is subsequently deallocated, the hook location will be cleared. This mechanism can be viewed as a means of recognizing those packets whose handles were once saved but which have been deallocated in the meantime and are not available any more.

```
---> void tcvp_unhook (address packet)
```

This operation removes the hook from the packet. The hook location is not affected.

```
---> void tcvp_settimer (address packet, word delay)
```

This function sets the timer for the indicated packet to go off after `delay` seconds. When the timer goes off, the `tcv_tmt` plugin function will be called for the packet. It is legal to set timers for packets waiting for reception or transmission. Nothing wrong will happen, if a packet is deallocated before its timer goes off.

```
---> void tcvp_cleartimer (address packet)
```

This function clears (unsets) the timer for the packet. Nothing happens if no timer was set.

```
---> int tcvp_control (int phid, int option, address value)
```

This is the plugin-callable function for accessing the control operation of a physical module by its numerical identifier. It provides a direct link to the control function registered by the respective physical module.

5. Library modules

The library (directory `Lib`) contains two sample physical interface modules and one skeletal plugin. One of the two physical modules (files `phys_ether.h` and `phys_ether.c`) interfaces TCV to the ETHERNET device, the other (files `phys_uart.h` and `phys_uart.c`) provides an interface to the UARTs.

→ phys_ether

An application that wants to use the ETHERNET physical module should include the file `phys_ether.h`, which declares the following configuration function:

```
void phys_ether (int id, int cid, int mbs)
```

that must be called to configure the module. The role of this function is to initialize the module and invoke `tcvphy_reg` (see above) to register the module with TCV. The arguments have the following meaning:

`id` is the numerical identifier to be assigned to the module as explained in section 2.

`cid` is the card identifier to be used by the ETHERNET driver (see [PicOS]).

`mbs` is the maximum buffer size used to determine the length of the reception buffer that the module must allocate to accommodate a received packet.

If `cid` is zero, the card identifier of the `ETHERNET` interface is left intact and the interface is set to operate in the raw mode, which means that the transmitted and received packets will be complete MAC level Ethernet frames [PicOS]. It is the responsibility of the plugin (or possibly the application) to make sure that those packets have the required appearance, including their MAC headers.[2] If `cid` is nonzero, the card identifier is set to `cid` and the `ETHERNET` interface is set to operate in the cooked mode, which means that TCV will only perceive the cooked payloads of the incoming and outgoing frames.

It is possible to initialize the module twice, using different numerical identifiers, once with `cid` equal zero, and once with a nonzero `cid`. This will effectively register two different physical modules (raw and cooked) sharing the same `ETHERNET` device in two different modes. The received frames recognized as cooked will have their

[2] Note that the sender address in an outgoing frame is inserted automatically by the ETHERNET driver.

payloads extracted and passed to TCV by the *cooked* module, while the raw frames will be delivered verbatim by the *raw* module. Outgoing frames submitted via the respective module instances will be handled in a similar manner. This way, as perceived by TCV and the plugins, the `ETHERNET` interface will essentially appear as two different devices.

The control function registered by the `ETHERNET` module offers the following operations (the symbolic constants represent the values of the `option` argument):

`PHYSOPT_STATUS`

The operation returns (via the function value as well as via the second argument) the status of the transmitter/receiver. Bit number 0 (from the right) is set if the receiver is on, and bit number 1 is set if the transmitter is on.

`PHYSOPT_TXON`

The operation switches on the transmitter. The transmitter is switched on by default when the module is initialized.

`PHYSOPT_TXOFF`

The operation switches off the transmitter. The transmitter remains active until the output queue of packets becomes empty. Then, subsequent packets arriving for transmission will be dropped until the transmitter is switched on again. The dropped packets will be treated as if they have been transmitted and will be perceived as such by the plugin.

`PHYSOPT_TXHOLD`

The transmitter is switched off, but the packets arriving for transmission are queued. They will be processed when the transmitter is switched on again.

`PHYSOPT_RXON`

The operation switches on the receiver. The receiver is switched on by default when the module is initialized.

`PHYSOPT_RXOFF`

The operation switches off the receiver.

`PHYSOPT_ERROR`

The operation returns (via the function value) the error status of the last erroneously received frame and clears the error status. See the description of the `ETHERNET` device in [PicOS] for more details.

All the above operations ignore the value argument. The first four of them can be viewed as standard and applicable to any physical module. In particular, they are also available for the UART and RADIO modules described below. Note that in addition to the control operation of TCV, the application/plugin can directly perform CONTROL requests (via io) on the ETHERNET device, if it knows the mapping of the physical module.

The *information* attribute of the phys_ether module is equal to 0x02v0, where v is 1 if the interface is *cooked* and 0 otherwise.

→ phys_uart

An application that wants to use this module should include phys_uart.h, which declares the following configuration function:

```
void phys_uart (int id, int uart, int mode, int mbs)
```

where id is the module identifier, uart indicates the UART device (UART_A or UART_B), mode selects one of two modes: UART_PHYS_MODE_EMU or UART_PHYS_MODE_DIRECT, and mbs, as in the previous case, specifies the size of the reception buffer (the maximum length of a received packet).

In both modes, packets sent and received over the UART are encapsulated into physical frames which are completely transparent to TCV, i.e., only the physical module is aware of their existence. As viewed by TCV, the packets handled by the module are arbitrary sequences of bytes (similar to cooked payloads of the ETHERNET interface). Of course, the plugin/application can impose some framing within those packets, which in turn is of no concern to the physical module.

The physical framing consists of a starting sequence at the beginning of a packet and an ending sequence at its end. The starting sequence is an arbitrary number of ASCII SYN characters (code 0x16) followed by DLE (0x10) and STX (0x02). The payload begins with the first character following STX. The ending sequence is DLE ETX (0x10 0x03). A DLE character occurring within the payload is escaped with another DLE. This escape is inserted by the module when the packet is transmitted, and stripped off when the packet is received. In fact, having received a DLE character that is not followed by another DLE, the module blindly assumes that the following character is ETX and terminates the packet. This is because a DLE character within the physical payload must be escaped to be considered part of the payload.

The length of the initial SYN sequence in a received frame is arbitrary. While looking for a packet in the received stream of characters, the module skips everything until it recognizes the sequence SYN DLE STX. In an outgoing frame sent by the module, the header consists of four SYN characters.

As the framing bytes are never actually *received* (meaning stored in the reception buffer), the mbs argument refers to the maximum size of the physical payload, excluding the framing (and possibly escaping) characters.

If mode is 0 (UART_PHYS_MODE_DIRECT), the operation is simple and straightforward. The UART is unlocked and all io operations are carried out in a blocking, interrupt-driven fashion. Outgoing packets are simply written to the UART (with the framing described above), while incoming packets are simply read from the device.

If mode is 1 (UART_PHYS_MODE_EMU), the module treats the UART as an approximation of a dumb radio channel. The UART is locked (which means that all io operations are carried out persistently without interrupts) and the operation of the transmitter/receiver is driven by delays. In particular, the receiver periodically attempts to receive a packet by persistently polling the UART, and if nothing shows up within a timeout, the attempt fails. Before sending a packet, the transmitter polls the UART for an activity and backs off randomly when an activity is recognized. The timing of this behavior is controlled by six parameters settable via the control function registered by the module.

It is possible to register two different physical modules for each of the two UARTs, with both modules operating in the same or different modes.

In addition to the *standard* control operations discussed for phys_ether, the UART module provides the following requests (all of them are void if mode is 1):

PHYSOPT_CAV

The value argument points to an unsigned integer number specifying the CAV (collision avoidance vector), i.e., the minimum delay until the next transmission attempt (in milliseconds). This delay will not be obeyed if the first outgoing packet is urgent, unless there are other reasons for deferring the transmission (e.g., an incoming activity perceived on the UART).

PHYSOPT_SENSE

The operation senses the UART by persistently trying to receive a byte for the amount of time equal to the current setting of the *sense interval* (see below). It returns 0 (via the function value) if nothing has been sensed during that time, and 1 otherwise.

PHYSOPT_SETPARAM

The value argument points to an array consisting of 2 words. The first word is a number from 0 to 5 indicating the parameter whose value should be changed, the second word specifies the new value of the parameter. Those parameters are listed below in the increasing order of their indexes:

delmnrcv(0) This is the minimum delay separating two receive attempts expressed in milliseconds.[3] The minimum is 1, the maximum is 32767, and the default is 2.

[3] A millisecond in PicOS is 1/1024 of a second.

`delmxrcv(1)` This is the maximum delay separating two receive attempts expressed in milliseconds. The actual delay is always between `delmnrcv` and `delmxrcv` inclusively. It starts at `delmnrcv` and after every failed receive attempt (no activity in the port) is incremented by one millisecond, but never above `delmxrcv`. It is reset to `delmnrcv` after any activity is detected in the UART. The minimum is 1, the maximum is 32767, the default is 16. This value is guaranteed to be not less than `delmnrcv`.

`delmnbkf(2)` This is the minimum amount of backoff (in milliseconds) used by the transmitter after the end of an activity perceived by the receiver in the port. The minimum is 0, the maximum is 32768, and the default is 8.

`delbsbkf(3)` This is the number of ones counting from the right in a bit mask determining the range of the extra delay added to `delmnbkf` to produce the actual (randomized) backoff value. The actual backoff is generated as `delmnbkf + (random & mask)`, where `random` is a pseudo random word-sized integer number. The minimum is zero, the maximum is 15, the default is 5 (which corresponds to the mask of `0x1f`).

`delxmsen(4)` This is the sense delay in milliseconds, i.e., the amount of time during which the transmitter senses the port before every transmission. If an activity is detected during that time, the transmission is postponed according to the backoff. If that activity turns out to be a valid packet, the packet is received. The minimum is 0, the maximum is 1024, and the default is 2.

`deltmrcv(5)` This is the receiver persistence in milliseconds, i.e., how long the receiver is going to wait for the first character of a packet before giving up. The minimum is 1, the maximum is 4096, and the default is 10.

By requesting a CAV setting, the application/plugin explicitly sets the current backoff delay (for the next transmission) to the maximum of its current setting and the requested CAV value. The backoff delay is always ignored if the top packet in the outgoing queue is urgent, unless the pre-transmission sensing detects an inbound activity.

Two additional timeout parameters are hardwired into the module. They can be changed by resetting the following symbolic constants declared in `Lib/phys_uart.h`:

UART_CHAR_TIMEOUT	(set at 5 milliseconds) gives the maximum waiting time for the next character of a packet. The packet is assumed to have terminated (prematurely) if the next character doesn't show up within that time.
UART_PACKET_SPACE	(set at 6 milliseconds) gives the minimum amount of space inserted between two consecutively transmitted packets.

We emphasize it once again that the delay parameters are ignored if the module was initialized with `mode` equal 0 (`UART_PHYS_MODE_DIRECT`), in which case the UART is treated as a straightforward, non-interfering, interrupt-driven device.

The *information* attribute of `phys_uart` is equal to `0x01vu`, where `v` is the interface mode (0/1) and `u` is the UART device number (0/1).

→ phys_radio

This module interfaces TCV to the raw `RADIO` driver. To use it, the application should include `phys_radio.h` providing the following configuration function:

```
void phys_radio (int phy, int mod, int mbs)
```

As usual, the first argument is the interface number. If the second argument is nonzero, it declares the station Id and selects the so-called framed reception mode, whereby the first word (two bytes) of the raw packet is expected to include a station Id. For a received packet, this word must match the station Id or be zero; otherwise, the packet will be ignored. However, for a transmitted packet, the station Id will not be automatically inserted, so it is up to the application/plugin to take care of this end.

The last argument indicates the maximum length of a legitimate packet, including the 4-byte checksum used by the `RADIO` driver (see [PicOS]).

A received and acceptable packet is passed directly to TCV along with its checksum and the possible station Id. A packet acquired for transmission, must have the station Id set (if relevant) and must provide room for the checksum.

The module requires that the `RADIO` device be configured with `RADIO_INTERRUPTS != 0` (see [PicOS]).

In addition to the *standard* control operations discussed for `phys_ether`, `phys_radio` provides the following requests:

`PHYSOPT_CAV`

The `value` argument points to an unsigned integer number specifying the CAV (collision avoidance vector), i.e., the minimum delay until the next transmission attempt (in milliseconds). This delay will not be obeyed if the first outgoing

packet is urgent, unless there are other reasons for deferring the transmission (e.g., an activity perceived by the receiver, see below).

`PHYSOPT_SENSE`

The operation returns the number of milliseconds that have elapsed since the last time an activity was sensed in the ether (or `MAX_INT` if that time is longer than that `MAX_INT` milliseconds). See [PicOS] for a discussion how the RADIO driver keeps track of this interval.

`PHYSOPT_SETPARAM`

The `value` argument points to an array consisting of 2 words. The first word is a number from 0 to 5 indicating the parameter whose value should be changed; the second word specifies the new value of the parameter. Those parameters are listed below in the increasing order of their indexes:

`delmnbkf(0)` This is the minimum amount of backoff (in milliseconds) used by the transmitter after an activity perceived by the receiver. The minimum is 1, the maximum is 32768, and the default is 32. `delmnbkf` is never less than `delxmsen` (see below).

`delbsbkf(1)` This is the number of ones counting from the right in a bit mask determining the range of the extra delay added to `delmnbkf` to produce the actual (randomized) backoff value. The actual backoff is generated as `delmnbkf + (random & mask)`, where `random` is a pseudo random word-sized integer number. The minimum is zero, the maximum is 15, the default is 8 (which corresponds to the mask of 0xff).

`delxmsen(2)` This is the minimum amount of time in milliseconds during which the ether should be silent before a transmission is allowed to commence. The minimum is 0, the maximum is 1024, and the default is 4.

`prlength(3)` This is the length of a transmitted packet preamble as the number of full high-low cycles. The minimum is 2, the maximum is 128, and the default is 32 or 24 depending on the radio board (see [PicOS]).

`prwaittm(4)` This is the maximum waiting time for a preamble before the `READ` (receive) operation is considered a failure (see [PicOS]). The minimum is 1, the maximum is 64, and the default is 4.

`prntries(5)` This is the maximum number of preamble resync attempts (see [PicOS]). The minimum is 1, the maximum is 8, and the default is 4.

By requesting a CAV setting, the application/plugin explicitly sets the current backoff delay (for the next transmission) to the maximum of its current setting and the requested CAV value. The backoff delay is always ignored if the top packet in the outgoing queue is urgent, unless an activity has been perceived in the ether less than `delxmsen` milliseconds ago.

The *information* attribute of `phys_radio` is equal to `0x030v`, where `v` is 1 with the framing option and 0 otherwise.

→ plug_null

The library includes a sample plugin whose role is to pass all incoming packets to the application and send out all packets created by the application. This plugin is very simple and can be viewed as the minimal piece of code that makes TCV operational. It can also be used as a skeleton for creating more sophisticated plugins.

To use this plugin, the application should include the file `plug_null.h`. For illustration, let us consider the sample application named `SerTest` whose purpose is to test `phys_uart` in a ping-like setup with another machine connected to the evaluation board via one of the two UARTs. In the initial state of its root process, this application executes the following code:

```
...
   #define    CHANNEL          UART_B
   #define    CHANNEL_MODE     UART_PHYS_MODE_DIRECT
   #define    MAXPLEN          256
...
   phys_uart (0, CHANNEL, CHANNEL_MODE, MAXPLEN);
   tcv_plug (0, &plug_null);
   sfd = tcv_open (NONE, 0, 0);
   if (sfd < 0) {
      diag ("Cannot open tcv interface");
      halt ();
   }
   tcv_control (sfd, PHYSOPT_TXOFF, NULL);
   tcv_control (sfd, PHYSOPT_RXOFF, NULL);
...
```

The second argument of `tcv_plug` is a pointer to the plugin structure (of type `tcvplug_t`) specifying the plugin functions. The `open` operation can never block, so its `state` argument is irrelevant.

When we look into `plug_null.c`, we see these definitions:

```
static int tcv_ope (int, int, va_list);
static int tcv_clo (int, int);
static int tcv_rcv (int, address, int, int*, tcvadp_t*);
static int tcv_frm (address, int, tcvadp_t*);
static int tcv_out (address);
static int tcv_xmt (address);

const tcvplug_t plug_null =
   {tcv_ope, tcv_clo, tcv_rcv, tcv_frm, tcv_out, tcv_xmt,
   NULL};
```

The last function of the plugin (responsible for handling timeouts) is absent (its pointer is `NULL`) as the plugin never uses the timers provided by TCV and, consequently, that function is never called. The most complex function of the plugin is `tcv_ope` listed below.

```
int *desc = NULL;

static int tcv_ope (int phy, int fd, va_list plid) {
/*
 * This is very simple - we are allowed to have one
 * descriptor per phy.
 */
   int i;

   if (desc == NULL) {
      desc = (int*) umalloc (sizeof (int) * TCV_MAX_PHYS);
      if (desc == NULL)
         syserror (EMALLOC, "plug_null tcv_ope");
      for (i = 0; i < TCV_MAX_PHYS; i++)
         desc [i] = NONE;
   }

   /* phy has been verified by TCV */
   if (desc [phy] != NONE)
      return ERROR;
      desc [phy] = fd;
      return 0;
   }
```

The function creates an array indexed by physical interfaces and allows the application to open one session (descriptor) per interface. Note that the values of the session descriptors are determined by TCV before `tcv_ope` is called. No extra open arguments are used by the plugin. The matching close operation looks like this:

```
static int tcv_clo (int phy, int fd) {

   /* phy/fd has been verified */
```

```
    if (desc == NULL || desc [phy] != fd)
        return ERROR;

    desc [phy] = NONE;
    return 0;
}
```

It just removes the session descriptor from the array. The least trivial among the remaining functions is this one:

```
static int tcv_rcv (int phy, address p, int len, int *ses,
tcvadp_t *b) {

    if (desc == NULL || (*ses = desc [phy]) == NONE)
        return TCV_DSP_PASS;

    b->head = b->tail = 0;

    return TCV_DSP_RCV;
}
```

which is responsible for claiming packets received by physical modules. It checks if there exists an open session whose physical module identifier matches the physical module that has received the packet. If this is the case, the packet is claimed and received. Its disposition is `TCV_DSP_RCV`, which means that the packet is put at the end of the session's incoming queue. Otherwise, the packet is not claimed by the plugin. The plugin sets both offsets determining the portion of the received string to be turned into the packet to zero, which results in passing the entire received string to TCV. These are the remaining functions of the plugin:

```
static int tcv_frm (address p, int phy, tcvadp_t *b) {

    return b->head = b->tail = 0;
}

static int tcv_out (address p) {

    return TCV_DSP_XMT;
}

static int tcv_xmt (address p) {

    return TCV_DSP_DROP;
}
```

Their semantics are trivial and obvious.

The *information* attribute of `plug_null` is `0x0001`.