VNETI
Versatile NETwork Interface

Version 3.0
December 2015

Preamble

The purpose of VNETI (which stands for Versatile NETwork Interface) is to provide a simple collection of API, independent of the underlying implementation of networking, which, in addition to enabling rapid deployment of wireless networked application based on microcontrollers 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.

1. The structure

The structure of VNETI and its relationship to 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.

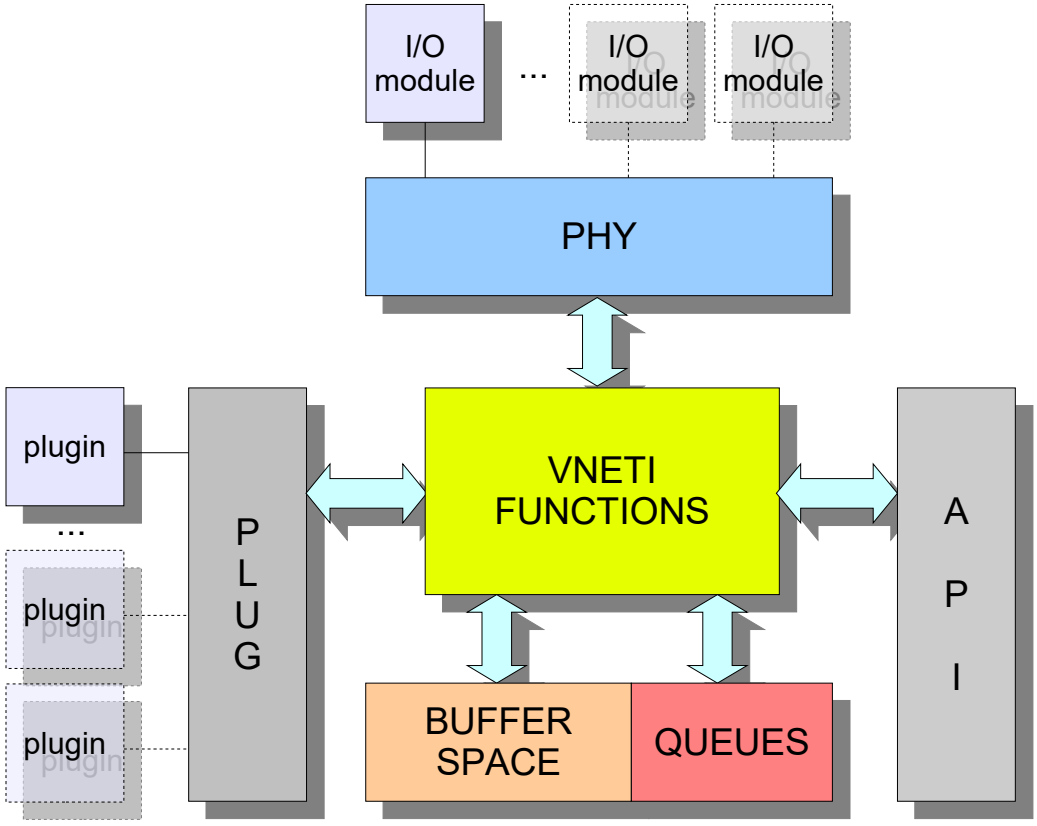

Figure 1: VNETI structure.

A workable VNETI setup involves at least one physical I/O module (PHY) 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. Regardless of the actual configuration of plugins and physical modules configured into the system, the API provided by VNETI is essentially fixed; thus, we can say that VNETI, 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.

2. Physical interface

Physical I/O modules (typically, but not necessarily, RF devices) are interfaced to VNETI via a collection of functions whose headers are defined in *tcvphys.h* (in directory PicOS). This interface is referred to as *PHY*. During initialization, a physical module registers itself with VNETI by executing this function:

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

whose first argument is a small integer number uniquely identifying the module within the current praxis.[1] The second argument points to the module's *control function* for setting various operating parameters of the module. The third argument provides the so-called interface *information* attribute i.e., an integer value that is supposed to be globally unique among all possible interfaces, and whose role is to identify the module unambiguously (regardless of its short and possibly ambiguous **id**) for the plugin or application.

The short module identifier (**id**) will allow a plugin to recognize one of possibly several physical modules, e.g., as the source of an inbound packet. Also, by passing it as a parameter to the **open** function (see below), the praxis will be able to associate a specific session with a specific physical module. Module identifiers 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** with the default value of 3 and the maximum of 8).

The control function pointed to by **cfun** 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 interface-specific (e.g., switching on the promiscuous mode of the Ethernet interface). 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 VNETI API gives the praxis access to the control functions of the physical modules handling its connections. If the program decides to take advantage of an exotic option, it is expected to know what it is doing.

A registered physical module is assigned a single queue of outgoing packets, which is handled automatically by VNETI. The value returned by **tcvphy_reg** is a 16-bit integer number identifying the *queue event* that will be triggered by VNETI 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)
```

[1] This is PicOS's term for “application.”

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 removes the first packet from the queue and returns its pointer, while the length of this packet (in bytes) is returned via the second argument. Note that packets (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 function of the physical module, e.g., after receiving the queue event.

The queue event is triggered whenever the outgoing queue of packets changes is status from empty to nonempty.

It is possible to peek at the topmost packet in the outgoing queue (without removing it) by calling this function:

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

with the module identifier as the argument. The function returns the pointer to the first packet in the queue, or **NULL** if the queue is empty.

The packet whose pointer is returned by **tcvphy_get** is stored in the dynamic buffer pool handled by VNETI. This means that the physical module must tell VNETI 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:

```
void tcvphy_end (address packet)
```

which accepts the packet pointer that was previously produced by **tcvphy_get** and notifies VNETI that the packet has been transmitted. This does not automatically result in the deallocation of the packet's storage. Determining the fate of the packet that has been transmitted by the physical module is up to the plugin, and **tcvphy_end** will consult a plugin function (**tcv_xmt** – see section 4) to decide what to do with the packet next. For example, the plugin may decide to keep the packet for a possible retransmission until it is acknowledged by the recipient.

A packet indicated by **tcvphy_top**, i.e., one that has not been removed from the outgoing queue, can also be marked as "transmitted" by **tcvphy_end**. If the plugin decides to deallocate such a packet, it will be automatically dequeued before the deletion.

When the physical module receives a packet that should be passed to VNETI, it calls the following function:

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

whose last two arguments describe the location of the packet in memory. Note that the physical module is responsible for allocating and handling its own reception buffer (pointed to by **buf**). The above operation never blocks and it expects that VNETI is able to copy the packet to its internal buffer storage, furnishing it with internal attributes based on the decision of the plugin. If no memory is available to accommodate the packet into a queue, it will be dropped and ignored, in which case the function returns zero. This will also happen if the plugin decides to drop the packet (e.g., considered irrelevant); thus, a zero value does not necessarily mean memory problems. If the packet has been stored for reception by the praxis (has not been dropped), the function returns 1.

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

This operation drains the output queue of the interface module and returns the number of erased packets. The packets are removed from the queue and deallocated without notifying the plugin.

3. API: the praxis interface

Before the praxis can use the services offered by VNETI (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 trivial *null* plugin (see file **plug_null.c** in *Apps/AppLib/PlugNull/*), there can only 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 some kind of connection to a 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 re-try the operation. The state argument can be **WNONE**, in which case the operation never blocks, but returns its status. If the value returned by **tcv_open** is −2 (constant **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 VNETI (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 invoking FSM is automatically blocked and restarted at the indicated state when the session status has changed.[2]

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 interfaced to VNETI by the praxis.

Upon a successful completion, **tcv_open** returns a small integer value (starting from zero), which identifies the session resembling a file descriptor (in UNIX terminology). If the value returned by the function is −1 (constant **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 8. 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.

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

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

[2] So far we haven't encountered a single case where blocking **tcv_open** would be useful. Consequently, the section of code implementing this feature has been made conditional on the compilation constant **TCV_OPEN_CAN_BLOCK** which must be set to 1 for this feature to be available.

whose second argument is a session descriptor[3] that was returned by **tcv_open**. The operation can legitimately block,[4] 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 VNETI. 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 determined by the plugin.

The following operation acquires the next packet queued for input 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). The operation will 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 contents, represents in fact a more elaborate data structure maintained by VNETI 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 FSMs). 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. The exact behavior of **tcv_endp** depends on the packet's characteristics. For an incoming packet, whose handle has been acquired by **tcv_rnp**, the function assumes that the application is completely done with the packet which should now be deallocated.

Outgoing packets are handled in a similar way, i.e., by requesting a packet handle from VNETI. This is done by calling this function:

```
address tcv_wnps (word state, int fd, int length, Boolean urg)
```

The operation builds a new outgoing packet **length** bytes long and returns its handle. It may block on account of the plugin (see section 4), or because there isn't enough free memory to accommodate the new packet. When that happens, the invoking process will be resumed in the indicated state when there is a new opportunity to try, i.e., the plugin triggers an event, or some memory becomes available. As usual, if **state** is **NONE**, the function never blocks but returns **NULL** on failure.

[3] In all subsequent occurrences, **fd** will stand for a session descriptor.
[4] The availability of this feature is conditional on the same constant as blocking **tcv_open**.

The last argument is the *urgent* flag, i.e., if *true* (or **YES** in our parlance) it says that the packet is "urgent". The standard interpretation of urgency of an outgoing packet (which can be augmented or even overridden by the plugin) is to push the packet to the front of the output queue of the respective PHY module (needless to say that normal, non-urgent, packets are appended at the end).

These two macros provide alternative ways of referencing the function:

```
#define tcv_wnp(s,f,l)  tcv_wnps (s, f, l, NO)
#define tcv_wnpu(s,f,l) tcv_wnps (s, f, l, YES)
```

The function builds a new packet (or rather a packet placeholder) with the intention of sending it out. While **tcv_rnp** should be viewed as an input (read) function, **tcv_wnps** is an output function, at least in the sense that the first function brings in an existing packet, while the second one creates a packet placeholder to be filled out by the caller. At this stage we can only talk about intentions, because the precise interpretation of those intentions is determined by the interaction of the plugin and the PHY module.

When the packet holder produced by **tcv_wnps** has been filled out and complete, the caller can use **tcv_endp** (see above) to notify VNETI that the new packet should be accepted as a ready outgoing packet. Note that VNETI is able to tell the difference between the two invocations of **tcv_endp** (i.e., for an incoming and an outgoing packet). This is possible because the complete data structure encompassing the packet stores its "intention", which is different for a received packet (one acquired by **tcv_rnp**) and for an outgoing packet (one returned by **tcv_wnps**).

Packets handled by **tcv_rnp**/**tcv_wnps** are complete in terms of their overall size, i.e., they include room for the possible headers and/or trailers. While those headers and trailers need not be used by the praxis (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 the handles. If the headers/trailers are processed by the plugin, the **length** parameter of **tcv_wnps** refers to the *logical* component of the packet (the application payload) excluding the parts that are not meant for the praxis. VNETI 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 bypass the header explicitly.

This may sound confusing, but the confusion is largely apparent and results from the open nature of the plugin/PHY interface. The plugin has a choice between exposing all packets to the application and marking some of their initial and/or trailing parts as "internal" (see section 4). In the second case, the concept of *payload length* comes into play. For example, the **length** argument of **tcv_wnps** refers to the payload length: when allocating memory for the packet, the plugin will augment the specified length with the length of the header and trailer. Note however, that the pointer returned by **tcv_rnp** or **tcv_wnps** always refers to the absolute beginning of the packet, as it is going to appear to the PHY module. Consequently, if the praxis wants to access the packet contents directly, it has to know how to skip the header. This function (in fact it is a macro) makes this operation transparent with respect to the header size (which need not be known by the praxis):

```
word tcv_offset (address packet)
```

returns the byte offset of the first payload byte of the packet (both for incoming and outgoing packets). Here is a sample way of using that offset:

```
address packet;
```


```
…
packet = tcv_wnps (ST_RETRY, fd, 10, NO);
strcpy (((char*)packet) + tcv_offset (packet), "message 1");
```

to make sure that the packet is properly filled with the payload. Note that regardless how much room the plugin needs for the header and trailer, the requested packet size (10) refers to the payload only.

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

```
int tcv_read (address packet, byte *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 VNETI 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. 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 in using the following function:

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

whose semantics is 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 set is the following one, which can also be used to determine the (payload) 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_wnps**, and then it is decremented by **tcv_write**.

If you decide to use **tcv_read**/**tcv_write**, then you should remember that these functions affect the payload offset returned by **tcv_offset** (in the most natural way). For example, when you write (we mean **tcv_write**) *k* bytes into the packet (or read, i.e., **tcv_read**, *k* bytes from the packet), the value subsequently returned by **tcv_offset** will be incremented by *k*.

The urgent attribute of a packet (it only makes sense for an outgoing packet) can also be set explicitly with this operation (implemented as a macro):

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

At first sight, you might expect that the effect of this sequence:

```
p = tcv_wnps (SOME_STATE, fd, len, NO);
tcv_urgent (p);
…
```


```
tcv_endp (p);
```

would be the same as that of:

```
p = tcv_wnps (SOME_STATE, fd, len, YES);
…
tcv_endp (p);
```

but that is only the case if the plugin doesn't interpret the urgent attribute at the stage of handling the buffer acquisition by **tcv_wnps**. Then, by the time we get to the actual queuing of the packet (by **tcv_endp**) the issue has been settled, so the standard interpretation of the attribute will apply. If you don't know, it is better to avoid using **tcv_urgent** (I was inclined to remove the operation, but it is only a macro, so why bother).

Here is the way to tell whether a packet is urgent:

```
bool tcv_isurgent (address packet)
```

The last function is a predicate returning nonzero when the urgent attribute of the packet is set. Note that the attribute does not only apply to outgoing packets (where it is set by the praxis); the plugin may decide that an incoming packet is also urgent with the effect of moving it to the front of the session's input queue. The urgent attribute of such a packet will be set, which fact can be detected (if needed) with **tcv_isurgent**.

The praxis is able to monitor the size of packet queues with help of the following function:

```
int tcv_qsize (int fd, int disp)
```

where **disp** is the so-called disposition code (see section 4), which, in this case can have one of these four values:

TCV_DSP_RCV the function returns the total number of packets awaiting reception by the praxis. These are the packets that have been queued for input via **tcv_rnp**.

TCV_DSP_RCVU the function returns the number of urgent packets awaiting reception by the praxis.

TCV_DSP_XMT the function returns the total number of packets awaiting transmission by the physical module responsible for handling the output from this session.

TCV_DSP_XMTU the function returns the number of urgent packets awaiting transmission by the physical module.

It is also possible to empty the respective queues by calling:

```
int tcv_erase (int fd, int disp)
```

where **disp** is exactly as for **tcv_qsize**. For **TCV_DSP_RCVU** and **TCV_DSP_XMTU**, the function completely clears the respective queues, while for the remaining two values it erases all non-urgent packets. Note that there is no way to erase all urgent packets leaving the non-urgent ones in the queue. A single packet available via a handle can be erased (unconditionally removed from any queue it may be in) with this function:

```
void tcv_drop (address packet)
```

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

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

which function is 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 VNETI 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 plugins configured by the praxis, e.g., for reference with **tcv_open**. The second argument points to a data structure describing the plugin. The layout of that structure is discussed 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, 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**) provides the plugin's global *information* descriptor intended to be globally unique among all plugins.

The plugin functions may reference a number of operations offered by VNETI to be used exclusively by plugins (discussed further in this document). The plugin functions can also call those functions of VNETI that are intended for the application (section 3).

We start from describing 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 VNETI 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 signals an 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, VNETI 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 praxis-level open function) may block is when the plugin **tcv_ope** function decides to block.[5] By itself, VNETI 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 VNETI 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**), VNETI examines all the registered plugins calling their **tcv_rcv** functions in turn). 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 (no further plugins are tried). This way 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 the 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 unsigned 16-bit numbers (PicOS type **word**) which determine the portion of the received packet that should be extracted and stored by VNETI. The first of these numbers is interpreted as the offset from the beginning of the received packet, while the second one is viewed as the offset from the end. In particular, if the entire received packet should be passed

[5] Note that the VNETI code implementing blocking in **tcv_open** is conditionally compiled (see section 3).

to VNETI, 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 (akin to data-link layer framing). This can also be done by the physical module, which decides on the portion of the received packet that should be presented to VNETI. 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 VNETI buffer and equipped with the standard set of attributes. This is in contrast to the remaining four plugin functions, which accept packet handles. You should remember that operations like cloning packets, changing their attributes, assigning them to sessions, and so on, are only defined on packet handles.

Among the attributes of a standard packet container are:

- the session (praxis session ID)
- the ID of the plugin responsible for handling the packet; this is the value assigned to the plugin by the first argument of **`tcv_plug`** (see section 3)
- the ID of the PHY module responsible for handling the packet; this is the value assigned to the PHY by the first argument of **`tcvphy_reg`** (see section 2)
- the *urgent* flag
- the *queued* flag (telling whether the packet is queued anywhere)
- the *outgoing* flag (telling whether the packet has been built by a call to **`tcv_wnps`**, as opposed to having been received); this flag is used by **`tcv_endp`** to differentiate its processing of the packet

The concept of disposition code, as returned by **`tcv_rcv`**, is general and applicable to some other plugin functions and VNETI operations available to plugins. Here is the complete list of those codes:

`(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 (tagged). If the packet is urgent (meaning its urgent attribute is set), 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 (tagged). 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 tagged with a legitimate (opened) session.

→ **tcv_frm**

This function serves a dual purpose. In the simpler case, it is called by **tcv_rnp** with a packet handle as the first argument and is expected to return via the second argument the offsets of the packet's header and trailer, thus isolating the packet's payload for the praxis (see section 3). The second argument points to a data structure declared like this:

```
typedef struct { word head, tail; } tcvadp_t;
```

where **head** is the payload offset from the front of the packet, while **tail** is the offset from the end. Thus, for example, when the payload extends over the entire packet (as claimed by **tcv_rcv** – see above), both values should be zero.

Another situation when **tcv_frm** is consulted is when **tcv_wnps** is called to preallocate a packet buffer for a new outgoing packet. This case is more complicated because:

1. the packet pointer is not yet available; it is in fact the purpose of the whole operation to create a new packet buffer and produce a packet pointer

2. if the plugin sees any reason why the packet acquisition (i.e., the call to **tcv_wnps**) should block, then **tcv_frm** must make this decision

The second responsibility of **tcv_frm** comes from the economy of tools. The function is called at the right place and at the right moment to perform the two tasks (so it is not necessary to introduce another function into the plugin). First, the framing information (the two offsets) must be known before the packet buffer can be allocated (because the offsets must be added to the payload length to determined the complete length of the packet). Then, at the same stage, the operation can be easily blocked (if required by the plugin), because nothing undoable (or difficult to undo) has been accomplished yet.

If the first argument of **tcv_frm** is not **NULL**, the function knows that it has been called on behalf of **tcv_rnp**. In that case, it is allowed to have a look at the packet (including its internal attributes) to come up with the two offsets, and store them in

the structure pointed to by the second argument. When the packet pointer is **NULL**, the function knows that it has been invoked by **tcv_wnps** to perform the two loosely related tasks listed above.

As in that case the function has no packet to inspect (and no information to support its calculation of the offsets), VNETI tries to help by passing two values in those offsets, before they are overwritten by whatever **tcv_frm** decides to store in them. Thus, **head** contains the session descriptor and **tail** carries the value of the *urgent* flag (zero meaning **NO**, and 1 meaning **YES**). Those values may also help the function to decide whether the operation should be blocked or not. In the former case, the function should return the identifier of an event that will be triggered by the plugin when the blocking condition may have disappeared. In that case, there is no need to store anything in the offsets. Otherwise (no blocking), the function should fill in the offsets and return zero.

In the vast majority of cases, the operation of **tcv_frm** is extremely simple. Many plugins use fixed offsets for all packets (so they don't need to inspect the packet for their calculation) and never block **tcv_wnps** for any plugin-specific reasons. The function may still block on the lack of buffer space, but it needs no plugin support for that.

→ **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 only 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 those VNETI functions that are intended to be used solely 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 praxis level functions or macros.

```
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 not 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 VNETI 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, should it be desired. Note that any packet can be explicitly discarded by setting its disposition code to **TCV_DSP_DROP**, or by calling **tcv_drop** (section 3).

```
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 looks like the natural 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 may be also OK, e.g., if the packet is to be transmitted and then dropped.

```
Boolean tcvp_isqueued (address packet)
```

The function (macro) returns **YES** if and only if the packet appears in one of the *transit* queues of VNETI, i.e., its is queued for reception (at a session) or for transmission (at a PHY). One can also say that such a packet is scheduled for automated processing by VNETI, as opposed to being removed from the transit queues, which typically means

that the packet *is* currently being processed (e.g., by the PHY, by the plugin, or by the application).

```
Boolean tcvp_isurgent (address packet)
```

This is another name for the tcv_isurgent macro (see section 3).

```
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 implementing safe packet deallocation, i.e., recognizing those packets whose handles were once saved (by the plugin) but which have been deallocated in the meantime (e.g., asynchronously and behind the scenes) and are not available any more.

Note: the hook operations, namely **tcvp_hook**, **tcvp_unhook**, and **tcvp_gethook**, are only available if the compilation constant **TCV_HOOKS** is defined as 1. This is because they need additional storage in the packet container.

```
void tcvp_unhook (address packet)
```

This macro removes the hook from the packet and also clears (sets to **NULL**) the hook location. If the packet is not hooked, the operation is void.

```
address *tcvp_gethook (address packet)
```

This macro returns the pointer to the hook location of the packet, i.e., the address of the location where the packet's hook is stored. If the packet is not hooked, the returned value is **NULL**.

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

This function sets the timer for the indicated packet to go off after **delay** milliseconds.[6] 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. If a packet is deallocated before its timer goes off, nothing will happen, i.e., the timer will be automatically voided.

Note: the timer operations: **tcvp_settimer**, **tcvp_cleartimer**, **tcvp_issettimer**, are only available if the compilation constant **TCV_TIMERS** is defined as 1. This is because they need additional storage in the packet container.

```
void tcvp_cleartimer (address packet)
```

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

```
Boolean tcvp_issettimer (address packet)
```

This function (macro) returns **YES** if and only if a timer is running for the packet (in other words, the packet is present in the timer queue). Note that this does not preclude the packet's simultaneous presence in a transient queue, e.g., the PHY transmit queue or a session input queue.

[6] These are "PicOS milliseconds." One PicOS millisecond = 1/1024 s.

```
Boolean tcvp_isdetached (address packet)
```

This function returns **YES** if and only if the packet is completely detached, i.e., it isn't stored in any transit queue (so **tcvp_isqueued** would return **NO**) and it also isn't waiting for a timer (so **tcvp_issettimer** would return **NO** as well).

```
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.

The NULL plugin

For illustration, we present here the minimal plugin needed to make VNETI functional as a raw interface to the network. This plugin can be found in file *plug_null.c* in *Apps/AppLib/PlugNull/*. Its simple role is to pass all incoming packets to the application and send out all packets created by the application.

To use this plugin, the application should include the file *plug_null.h*. For illustration, let us consider the following application fragment, which registers a hypothetical physical module, then the NULL plugin, and finally opens a session.

```
...
   tcvphy_reg (0, ctrlfun, MODULE_INFO);
   tcv_plug (0, &plug_null);
   sfd = tcv_open (NONE, 0, 0);
   if (sfd < 0) {
      diag ("Cannot open tcv interface");
      halt ();
   }
   tcv_control (sfd, PHYSOPT_OFF, 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 in this case, 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, 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, 0x0001};
```

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 VNETI and, consequently, that function is never called. The most complex function of the plugin is **tcv_ope** listed below.

```
static int *desc = NULL;

static int tcv_ope (int phy, int fd, va_list plid) {
   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 VNETI 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) {
   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 VNETI. These are the remaining functions of the plugin:

```
static int tcv_frm (address p, 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.