ppp_generic.txt
PPP Generic Driver and Channel Interface

Paul Mackerras
paulus@samba.org
7 Feb 2002

The generic PPP driver in linux-2.4 provides an implementation of the
functionality which is of use in any PPP implementation, including:

* the network interface unit (ppp0 etc.)
* the interface to the networking code
* PPP multilink: splitting datagrams between multiple links, and
  ordering and combining received fragments
* the interface to pppd, via a /dev/ppp character device
* packet compression and decompression
* TCP/IP header compression and decompression
* detecting network traffic for demand dialling and for idle timeouts
* simple packet filtering

For sending and receiving PPP frames, the generic PPP driver calls on
the services of PPP `channels'. A PPP channel encapsulates a
mechanism for transporting PPP frames from one machine to another. A
PPP channel implementation can be arbitrarily complex internally but
has a very simple interface with the generic PPP code: it merely has
to be able to send PPP frames, receive PPP frames, and optionally
handle ioctl requests. Currently there are PPP channel
implementations for asynchronous serial ports, synchronous serial
ports, and for PPP over ethernet.

This architecture makes it possible to implement PPP multilink in a
natural and straightforward way, by allowing more than one channel to
be linked to each ppp network interface unit. The generic layer is
responsible for splitting datagrams on transmit and recombining them
on receive.

PPP channel API

See include/linux/ppp_channel.h for the declaration of the types and
functions used to communicate between the generic PPP layer and PPP
channels.

Each channel has to provide two functions to the generic PPP layer,
via the ppp_channel.ops pointer:

* start_xmit() is called by the generic layer when it has a frame to
  send. The channel has the option of rejecting the frame for
  flow-control reasons. In this case, start_xmit() should return 0
  and the channel should call the ppp_output_wakeup() function at a
  later time when it can accept frames again, and the generic layer
  will then attempt to retransmit the rejected frame(s). If the frame
  is accepted, the start_xmit() function should return 1.

* ioctl() provides an interface which can be used by a user-space

program to control aspects of the channel's behaviour. This
procedure will be called when a user-space program does an ioctl
system call on an instance of /dev/ppp which is bound to the
channel. (Usually it would only be pppd which would do this.)

The generic PPP layer provides seven functions to channels:

* ppp_register_channel() is called when a channel has been created, to
  notify the PPP generic layer of its presence. For example, setting
  a serial port to the PPPDISC line discipline causes the ppp_async
  channel code to call this function.

* ppp_unregister_channel() is called when a channel is to be
  destroyed. For example, the ppp_async channel code calls this when
  a hangup is detected on the serial port.

* ppp_output_wakeup() is called by a channel when it has previously
  rejected a call to its start_xmit function, and can now accept more
  packets.

* ppp_input() is called by a channel when it has received a complete
  PPP frame.

* ppp_input_error() is called by a channel when it has detected that a
  frame has been lost or dropped (for example, because of a FCS (frame
  check sequence) error).

* ppp_channel_index() returns the channel index assigned by the PPP
  generic layer to this channel. The channel should provide some way
  (e.g. an ioctl) to transmit this back to user-space, as user-space
  will need it to attach an instance of /dev/ppp to this channel.

* ppp_unit_number() returns the unit number of the ppp network
  interface to which this channel is connected, or -1 if the channel
  is not connected.

Connecting a channel to the ppp generic layer is initiated from the
channel code, rather than from the generic layer. The channel is
expected to have some way for a user-level process to control it
independently of the ppp generic layer. For example, with the
ppp_async channel, this is provided by the file descriptor to the
serial port.

Generally a user-level process will initialize the underlying
communications medium and prepare it to do PPP. For example, with an
async tty, this can involve setting the tty speed and modes, issuing
modem commands, and then going through some sort of dialog with the
remote system to invoke PPP service there. We refer to this process
as `discovery'. Then the user-level process tells the medium to
become a PPP channel and register itself with the generic PPP layer.
The channel then has to report the channel number assigned to it back
to the user-level process. From that point, the PPP negotiation code
in the PPP daemon (pppd) can take over and perform the PPP
negotiation, accessing the channel through the /dev/ppp interface.

At the interface to the PPP generic layer, PPP frames are stored in

skbuff structures and start with the two-byte PPP protocol number.
The frame does *not* include the 0xff `address' byte or the 0x03
`control' byte that are optionally used in async PPP. Nor is there
any escaping of control characters, nor are there any FCS or framing
characters included. That is all the responsibility of the channel
code, if it is needed for the particular medium. That is, the skbuffs
presented to the start_xmit() function contain only the 2-byte
protocol number and the data, and the skbuffs presented to ppp_input()
must be in the same format.

The channel must provide an instance of a ppp_channel struct to
represent the channel. The channel is free to use the `private' field
however it wishes. The channel should initialize the `mtu' and
`hdrlen' fields before calling ppp_register_channel() and not change
them until after ppp_unregister_channel() returns. The `mtu' field
represents the maximum size of the data part of the PPP frames, that
is, it does not include the 2-byte protocol number.

If the channel needs some headroom in the skbuffs presented to it for
transmission (i.e., some space free in the skbuff data area before the
start of the PPP frame), it should set the `hdrlen' field of the
ppp_channel struct to the amount of headroom required. The generic
PPP layer will attempt to provide that much headroom but the channel
should still check if there is sufficient headroom and copy the skbuff
if there isn't.

On the input side, channels should ideally provide at least 2 bytes of
headroom in the skbuffs presented to ppp_input(). The generic PPP
code does not require this but will be more efficient if this is done.

Buffering and flow control

The generic PPP layer has been designed to minimize the amount of data
that it buffers in the transmit direction. It maintains a queue of
transmit packets for the PPP unit (network interface device) plus a
queue of transmit packets for each attached channel. Normally the
transmit queue for the unit will contain at most one packet; the
exceptions are when pppd sends packets by writing to /dev/ppp, and
when the core networking code calls the generic layer's start_xmit()
function with the queue stopped, i.e. when the generic layer has
called netif_stop_queue(), which only happens on a transmit timeout.
The start_xmit function always accepts and queues the packet which it
is asked to transmit.

Transmit packets are dequeued from the PPP unit transmit queue and
then subjected to TCP/IP header compression and packet compression
(Deflate or BSD-Compress compression), as appropriate. After this
point the packets can no longer be reordered, as the decompression
algorithms rely on receiving compressed packets in the same order that
they were generated.

If multilink is not in use, this packet is then passed to the attached
channel's start_xmit() function. If the channel refuses to take
the packet, the generic layer saves it for later transmission. The

generic layer will call the channel's start_xmit() function again
when the channel calls ppp_output_wakeup() or when the core
networking code calls the generic layer's start_xmit() function
again. The generic layer contains no timeout and retransmission
logic; it relies on the core networking code for that.

If multilink is in use, the generic layer divides the packet into one
or more fragments and puts a multilink header on each fragment. It
decides how many fragments to use based on the length of the packet
and the number of channels which are potentially able to accept a
fragment at the moment. A channel is potentially able to accept a
fragment if it doesn't have any fragments currently queued up for it
to transmit. The channel may still refuse a fragment; in this case
the fragment is queued up for the channel to transmit later. This
scheme has the effect that more fragments are given to higher-
bandwidth channels. It also means that under light load, the generic
layer will tend to fragment large packets across all the channels,
thus reducing latency, while under heavy load, packets will tend to be
transmitted as single fragments, thus reducing the overhead of
fragmentation.

SMP safety

The PPP generic layer has been designed to be SMP-safe. Locks are
used around accesses to the internal data structures where necessary
to ensure their integrity. As part of this, the generic layer
requires that the channels adhere to certain requirements and in turn
provides certain guarantees to the channels. Essentially the channels
are required to provide the appropriate locking on the ppp_channel
structures that form the basis of the communication between the
channel and the generic layer. This is because the channel provides
the storage for the ppp_channel structure, and so the channel is
required to provide the guarantee that this storage exists and is
valid at the appropriate times.

The generic layer requires these guarantees from the channel:

* The ppp_channel object must exist from the time that
  ppp_register_channel() is called until after the call to
  ppp_unregister_channel() returns.

* No thread may be in a call to any of ppp_input(), ppp_input_error(),
  ppp_output_wakeup(), ppp_channel_index() or ppp_unit_number() for a
  channel at the time that ppp_unregister_channel() is called for that
  channel.

* ppp_register_channel() and ppp_unregister_channel() must be called
  from process context, not interrupt or softirq/BH context.

* The remaining generic layer functions may be called at softirq/BH
  level but must not be called from a hardware interrupt handler.

* The generic layer may call the channel start_xmit() function at
  softirq/BH level but will not call it at interrupt level. Thus the

   start_xmit() function may not block.

* The generic layer will only call the channel ioctl() function in
  process context.

The generic layer provides these guarantees to the channels:

* The generic layer will not call the start_xmit() function for a
  channel while any thread is already executing in that function for
  that channel.

* The generic layer will not call the ioctl() function for a channel
  while any thread is already executing in that function for that
  channel.

* By the time a call to ppp_unregister_channel() returns, no thread
  will be executing in a call from the generic layer to that channel's
  start_xmit() or ioctl() function, and the generic layer will not
  call either of those functions subsequently.

Interface to pppd

The PPP generic layer exports a character device interface called
/dev/ppp. This is used by pppd to control PPP interface units and
channels. Although there is only one /dev/ppp, each open instance of
/dev/ppp acts independently and can be attached either to a PPP unit
or a PPP channel. This is achieved using the file->private_data field
to point to a separate object for each open instance of /dev/ppp. In
this way an effect similar to Solaris' clone open is obtained,
allowing us to control an arbitrary number of PPP interfaces and
channels without having to fill up /dev with hundreds of device names.

When /dev/ppp is opened, a new instance is created which is initially
unattached. Using an ioctl call, it can then be attached to an
existing unit, attached to a newly-created unit, or attached to an
existing channel. An instance attached to a unit can be used to send
and receive PPP control frames, using the read() and write() system
calls, along with poll() if necessary. Similarly, an instance
attached to a channel can be used to send and receive PPP frames on
that channel.

In multilink terms, the unit represents the bundle, while the channels
represent the individual physical links. Thus, a PPP frame sent by a
write to the unit (i.e., to an instance of /dev/ppp attached to the
unit) will be subject to bundle-level compression and to fragmentation
across the individual links (if multilink is in use). In contrast, a
PPP frame sent by a write to the channel will be sent as-is on that
channel, without any multilink header.

A channel is not initially attached to any unit. In this state it can
be used for PPP negotiation but not for the transfer of data packets.
It can then be connected to a PPP unit with an ioctl call, which
makes it available to send and receive data packets for that unit.

The ioctl calls which are available on an instance of /dev/ppp depend
on whether it is unattached, attached to a PPP interface, or attached
to a PPP channel.   The ioctl calls which are available on an
unattached instance are:

* PPPIOCNEWUNIT creates a new PPP interface and makes this /dev/ppp
  instance the "owner" of the interface.   The argument should point to
  an int which is the desired unit number if >= 0, or -1 to assign the
  lowest unused unit number.   Being the owner of the interface means
  that the interface will be shut down if this instance of /dev/ppp is
  closed.

* PPPIOCATTACH attaches this instance to an existing PPP interface.
  The argument should point to an int containing the unit number.
  This does not make this instance the owner of the PPP interface.

* PPPIOCATTCHAN attaches this instance to an existing PPP channel.
  The argument should point to an int containing the channel number.

The ioctl calls available on an instance of /dev/ppp attached to a
channel are:

* PPPIOCDETACH detaches the instance from the channel.   This ioctl is
  deprecated since the same effect can be achieved by closing the
  instance.   In order to prevent possible races this ioctl will fail
  with an EINVAL error if more than one file descriptor refers to this
  instance (i.e. as a result of dup(), dup2() or fork()).

* PPPIOCCONNECT connects this channel to a PPP interface.   The
  argument should point to an int containing the interface unit
  number.   It will return an EINVAL error if the channel is already
  connected to an interface, or ENXIO if the requested interface does
  not exist.

* PPPIOCDISCONN disconnects this channel from the PPP interface that
  it is connected to.   It will return an EINVAL error if the channel
  is not connected to an interface.

* All other ioctl commands are passed to the channel ioctl() function.

The ioctl calls that are available on an instance that is attached to
an interface unit are:

* PPPIOCSMRU sets the MRU (maximum receive unit) for the interface.
  The argument should point to an int containing the new MRU value.

* PPPIOCSFLAGS sets flags which control the operation of the
  interface.   The argument should be a pointer to an int containing
  the new flags value.   The bits in the flags value that can be set
  are:
        SC_COMP_TCP   enable transmit TCP header compression
        SC_NO_TCP_CCID   disable connection-id compression for
                                TCP header compression
        SC_REJ_COMP_TCP   disable receive TCP header decompression
        SC_CCP_OPEN   Compression Control Protocol (CCP) is
                                open, so inspect CCP packets

```
           SC_CCP_UP                CCP is up, may (de)compress packets
           SC_LOOP_TRAFFIC          send IP traffic to pppd
           SC_MULTILINK             enable PPP multilink fragmentation on
                                    transmitted packets
           SC_MP_SHORTSEQ           expect short multilink sequence
                                    numbers on received multilink fragments
           SC_MP_XSHORTSEQ          transmit short multilink sequence nos.
```

The values of these flags are defined in <linux/if_ppp.h>. Note
that the values of the SC_MULTILINK, SC_MP_SHORTSEQ and
SC_MP_XSHORTSEQ bits are ignored if the CONFIG_PPP_MULTILINK option
is not selected.

* PPPIOCGFLAGS returns the value of the status/control flags for the
  interface unit. The argument should point to an int where the ioctl
  will store the flags value. As well as the values listed above for
  PPPIOCSFLAGS, the following bits may be set in the returned value:

```
           SC_COMP_RUN              CCP compressor is running
           SC_DECOMP_RUN            CCP decompressor is running
           SC_DC_ERROR              CCP decompressor detected non-fatal error
           SC_DC_FERROR             CCP decompressor detected fatal error
```

* PPPIOCSCOMPRESS sets the parameters for packet compression or
  decompression. The argument should point to a ppp_option_data
  structure (defined in <linux/if_ppp.h>), which contains a
  pointer/length pair which should describe a block of memory
  containing a CCP option specifying a compression method and its
  parameters. The ppp_option_data struct also contains a `transmit'
  field. If this is 0, the ioctl will affect the receive path,
  otherwise the transmit path.

* PPPIOCGUNIT returns, in the int pointed to by the argument, the unit
  number of this interface unit.

* PPPIOCSDEBUG sets the debug flags for the interface to the value in
  the int pointed to by the argument. Only the least significant bit
  is used; if this is 1 the generic layer will print some debug
  messages during its operation. This is only intended for debugging
  the generic PPP layer code; it is generally not helpful for working
  out why a PPP connection is failing.

* PPPIOCGDEBUG returns the debug flags for the interface in the int
  pointed to by the argument.

* PPPIOCGIDLE returns the time, in seconds, since the last data
  packets were sent and received. The argument should point to a
  ppp_idle structure (defined in <linux/ppp_defs.h>). If the
  CONFIG_PPP_FILTER option is enabled, the set of packets which reset
  the transmit and receive idle timers is restricted to those which
  pass the `active' packet filter.

* PPPIOCSMAXCID sets the maximum connection-ID parameter (and thus the
  number of connection slots) for the TCP header compressor and
  decompressor. The lower 16 bits of the int pointed to by the
  argument specify the maximum connection-ID for the compressor. If
  the upper 16 bits of that int are non-zero, they specify the maximum

connection-ID for the decompressor, otherwise the decompressor's maximum connection-ID is set to 15.

* PPPIOCSNPMODE sets the network-protocol mode for a given network protocol. The argument should point to an npioctl struct (defined in <linux/if_ppp.h>). The `protocol' field gives the PPP protocol number for the protocol to be affected, and the `mode' field specifies what to do with packets for that protocol:

>   NPMODE_PASS   normal operation, transmit and receive packets
>   NPMODE_DROP   silently drop packets for this protocol
>   NPMODE_ERROR   drop packets and return an error on transmit
>   NPMODE_QUEUE   queue up packets for transmit, drop received
>   packets

  At present NPMODE_ERROR and NPMODE_QUEUE have the same effect as NPMODE_DROP.

* PPPIOCGNPMODE returns the network-protocol mode for a given protocol. The argument should point to an npioctl struct with the `protocol' field set to the PPP protocol number for the protocol of interest.  On return the `mode' field will be set to the network-protocol mode for that protocol.

* PPPIOCSPASS and PPPIOCSACTIVE set the `pass' and `active' packet filters. These ioctls are only available if the CONFIG_PPP_FILTER option is selected. The argument should point to a sock_fprog structure (defined in <linux/filter.h>) containing the compiled BPF instructions for the filter. Packets are dropped if they fail the `pass' filter; otherwise, if they fail the `active' filter they are passed but they do not reset the transmit or receive idle timer.

* PPPIOCSMRRU enables or disables multilink processing for received packets and sets the multilink MRRU (maximum reconstructed receive unit). The argument should point to an int containing the new MRRU value. If the MRRU value is 0, processing of received multilink fragments is disabled. This ioctl is only available if the CONFIG_PPP_MULTILINK option is selected.

Last modified: 7-feb-2002