Linux ATM device driver interface
Draft, version 0.1*

Werner Almesberger
werner.almesberger@lrc.di.epfl.ch

Laboratoire de Réseaux de Communication (LRC)
EPFL, CH-1015 Lausanne, Switzerland

February 5, 1996

Contents

1 System overview **1**

2 Device data **4**

3 VC data **6**

4 Socket buffer extensions **7**

5 Device operations **8**

6 Protocol operations **10**

7 Physical layer device operations **10**

8 Support functions **11**

9 Summary **12**

1 System overview

Figure 1 illustrates the environment a Linux ATM device driver operates in. "Device-independent ATM coordination" is basically a set of common data structures and conventions. "Protocol" denotes whatever uses the ATM device driver in a given context. Currently, this is either a raw transport or IP over ATM. ATM socket layer and protocol are not clearly separated in the current implementation, so the distinction between them may occasionally be somewhat arbitrary.

An ATM device driver consists of two parts: a usually small driver for the physical layer unit (PHY) that controls physical layer operation and gathers statistics, and the actual driver which is responsible for controlling the segmentation and reassembly (SAR) process, resource allocation, and for coordination with the protocol, the PHY driver, and the hardware. The reason for splitting the SAR and the PHY

*This document describes the device driver interface of Linux ATM release 0.7. Other releases may differ to some extent.

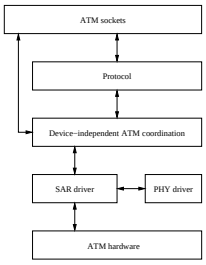

Figure 1: General ATM device access structure.

component is mainly that one could expect that several different PHY chips are used with the same SAR chip, and that the same PHY chip might actually appear on several adapters equipped with different SAR chips.

Normally, segmentation and reassembly and related functions are performed directly in hardware,[1] although a driver for very simplistic ATM adapters may also perform those operations in software. Similarly, traffic shaping is normally performed in hardware, but a driver may also do some or all of it in software.

Only the SAR driver needs to have knowledge of how the hardware is accessed. It provides an interface for accessing the PHY device to the PHY driver.

Some of the device-independent data structures are also manipulated by the socket layer. Note that some protocols (e.g. IP over ATM) may go through additional layers before data eventually reaches a socket (if at all).

Figure 2 gives a simplified view of the device-independent data structures. Many of the device-independent structures contain pointers to device-dependent structures which are under the control of the device driver.

`atm_dev` represents an ATM device. It contains pointers to device operations (see section 5), to operations on the PHY device (see section 7), to the private data structures of the SAR and the PHY drivers, the list of VC descriptors, and several sets of statistics counters (one set per AAL).

A VC descriptor (`atm_vcc`) contains all information pertaining to a VC, i.e. the VPI and VCI numbers, the AAL number, the address family, traffic parameters, etc. In addition to that, there are a few pointers: back to the device structure, to protocol functions (see section 6), to driver-private data, to protocol-private data, to the corresponding statistics block in the device structure, to the receive queue, and to the next and the previous VC descriptor.

The receive queue contains incoming SDUs waiting for final delivery by the ATM socket layer. Queues for data waiting for transmission are maintained by the device driver.

[1] Throughout this document "in hardware" means that the respective function is performed by the adapter.

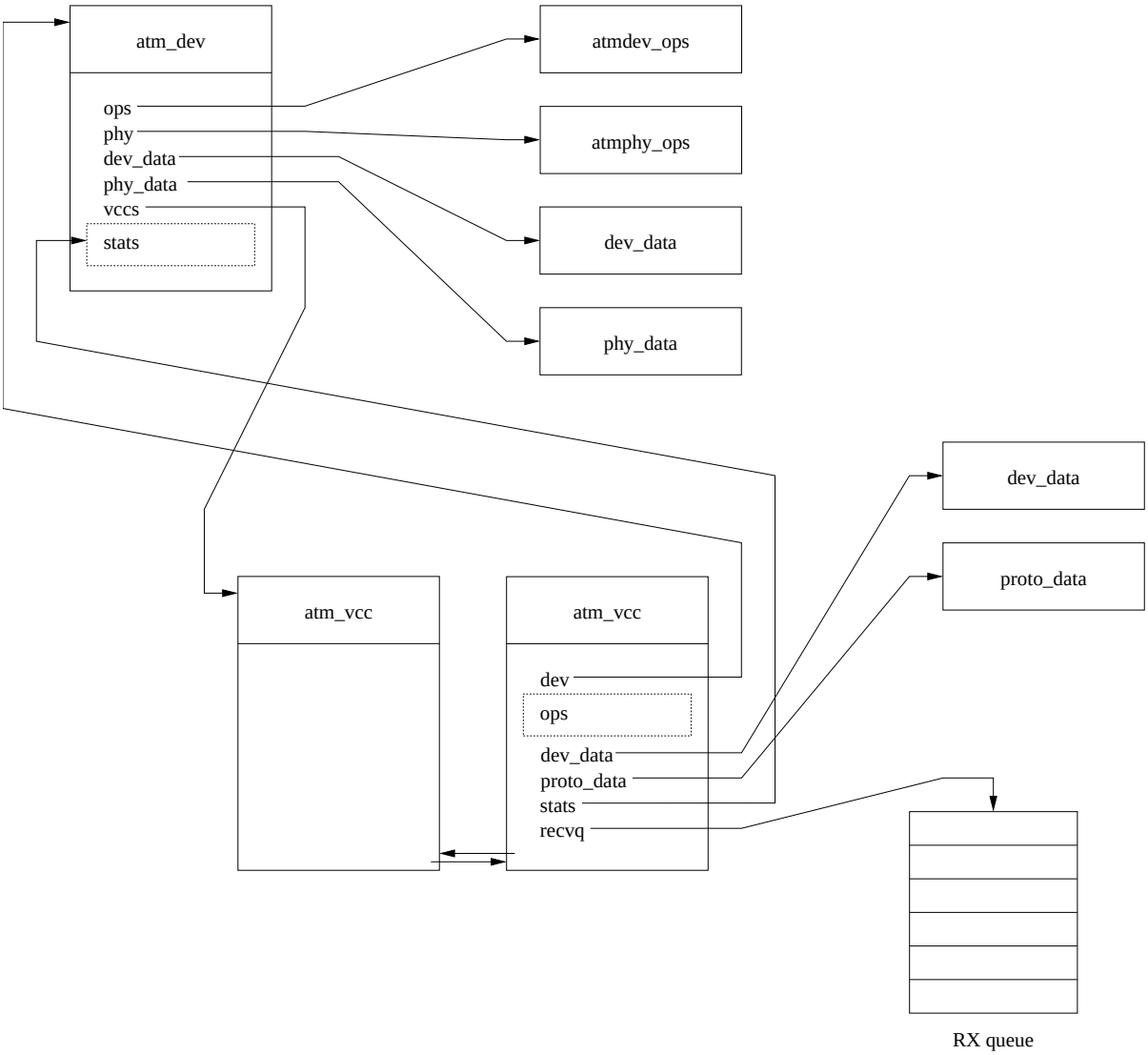

Figure 2: Device and VC data structures.

2 Device data

Device-independent ATM interface parameters are stored in `struct atm_dev`:

```
struct atm_dev {
        const struct atmdev_ops *ops;
        const struct atmphy_ops *phy;
        const char      *type;
        int             number;
        struct atm_vcc  *vccs;
        struct atm_vcc  *last;
        void            *dev_data;
        void            *phy_data;
        atm_dev_flags_t flags;
        struct atm_dev_addr *local;
        unsigned char   esi[ESI_LEN];
        struct atm_cirange ci_range;
        struct k_atm_dev_stats stats;
        char            signal;
        int             link_rate;
#ifdef CONFIG_PROC_FS
        struct proc_dir_entry *proc_entry;
        char *proc_name;
#endif
        struct atm_dev  *prev,*next;
};
```

`ops` points to the device operation function pointers, see also section 5. `ops` is set by `atm_dev_register` when registering the device.

`phy` points to PHY device operation function pointers, see also section 7. If there is no physical layer device or if it is directly controlled by the SAR driver, `phy` is not used and may be set to NULL. Otherwise, the PHY driver initialization function has to set `phy` when invoked by the SAR driver.

`type` points to the device type name. The device type is used to identify the driver in kernel messages and typically indicates the brand and the model number of the ATM adapter. `number` is the ATM interface number. `type` and `number` are set by `atm_dev_register`.

`vccs` points to the descriptor of the first VC of that device. If there are no VCs, `vccs` has to be set to NULL. It is initialized to NULL by `atm_dev_register`. `last` points to the last VC descriptor or is undefined if there is none. VCs are placed on the VC list in an arbitrary order.

`dev_data` and `phy_data` point to private data structures belonging to the SAR driver and the PHY driver, respectively. The content of these pointers is never accessed by any other part of the system than by the respective driver. If a driver does not wish to use private data structures, it can therefore leave the pointers uninitialized.

`local` points to the list of local ATM addresses of the interface. The same conventions as for the corresponding fields in SVC address structures apply (see also [2]). `local` can be set or changed at any time (except from interrupts). It is initialized to NULL by `atm_dev_register`.

`esi` contains the device's end system identifier (ESI). `esi` should be set during device initialization. If no ESI is available, the array should be filled with zero bytes.

`ci_range` specifies the range of VPIs and VCIs that are supported by that device. `ci_range` has to be initialized during device initialization and may be changed later by the device driver.

stats contains counters for AAL-level events. The counters are zero-initialized by atm_dev_register, maintained by the driver, and read and possibly reset (zeroed) by device-independent functions. Interrupts are disabled while reading or resetting the counters.

The counters are grouped in sets for each AAL type:

```
struct atm_dev_stats {
        struct atm_aal_stats aal0;
        struct atm_aal_stats aal34;
        struct atm_aal_stats aal5;
} __ATM_API_ALIGN;
#define ATM_GETLINKRATE _IOW('a',ATMIOC_ITF+1,struct atmif_sioc)
#define ATM_GETNAMES    _IOW('a',ATMIOC_ITF+3,struct atm_iobuf)
#define ATM_GETTYPE     _IOW('a',ATMIOC_ITF+4,struct atmif_sioc)
#define ATM_GETESI      _IOW('a',ATMIOC_ITF+5,struct atmif_sioc)
#define ATM_GETADDR     _IOW('a',ATMIOC_ITF+6,struct atmif_sioc)
#define ATM_RSTADDR     _IOW('a',ATMIOC_ITF+7,struct atmif_sioc)
#define ATM_ADDADDR     _IOW('a',ATMIOC_ITF+8,struct atmif_sioc)
#define ATM_DELADDR     _IOW('a',ATMIOC_ITF+9,struct atmif_sioc)
#define ATM_GETCIRANGE  _IOW('a',ATMIOC_ITF+10,struct atmif_sioc)
#define ATM_SETCIRANGE  _IOW('a',ATMIOC_ITF+11,struct atmif_sioc)
#define ATM_SETESI      _IOW('a',ATMIOC_ITF+12,struct atmif_sioc)
#define ATM_SETESIF     _IOW('a',ATMIOC_ITF+13,struct atmif_sioc)
#define ATM_GETSTAT     _IOW('a',ATMIOC_SARCOM+0,struct atmif_sioc)
#define ATM_GETSTATZ    _IOW('a',ATMIOC_SARCOM+1,struct atmif_sioc)
#define ATM_GETLOOP     _IOW('a',ATMIOC_SARCOM+2,struct atmif_sioc)
#define ATM_SETLOOP     _IOW('a',ATMIOC_SARCOM+3,struct atmif_sioc)
#define ATM_QUERYLOOP   _IOW('a',ATMIOC_SARCOM+4,struct atmif_sioc)
#define ATM_SETSC       _IOW('a',ATMIOC_SPECIAL+1,int)
#define ATM_ITFTYP_LEN  8
 * Loopback modes for ATM_{PHY,SAR}_{GET,SET}LOOP
 */
#define __ATM_LM_NONE   0
#define __ATM_LM_AAL    1
#define __ATM_LM_ATM    2
#define __ATM_LM_PHY    8
#define __ATM_LM_ANALOG 16
#define __ATM_LM_MKLOC(n)       ((n))
#define __ATM_LM_MKRMT(n)       ((n) << 8)
#define __ATM_LM_XTLOC(n)       ((n) & 0xff)
#define __ATM_LM_XTRMT(n)       (((n) >> 8) & 0xff)
#define ATM_LM_NONE     0
#define ATM_LM_LOC_AAL  __ATM_LM_MKLOC(__ATM_LM_AAL)
#define ATM_LM_LOC_ATM  __ATM_LM_MKLOC(__ATM_LM_ATM)
#define ATM_LM_LOC_PHY  __ATM_LM_MKLOC(__ATM_LM_PHY)
#define ATM_LM_LOC_ANALOG __ATM_LM_MKLOC(__ATM_LM_ANALOG)
#define ATM_LM_RMT_AAL  __ATM_LM_MKRMT(__ATM_LM_AAL)
#define ATM_LM_RMT_ATM  __ATM_LM_MKRMT(__ATM_LM_ATM)
#define ATM_LM_RMT_PHY  __ATM_LM_MKRMT(__ATM_LM_PHY)
#define ATM_LM_RMT_ANALOG __ATM_LM_MKRMT(__ATM_LM_ANALOG)
 * Note: ATM_LM_LOC_* and ATM_LM_RMT_* can be combined, provided that
 * __ATM_LM_XTLOC(x) <= __ATM_LM_XTRMT(x)
 */
```

```
struct atm_iobuf {
        int length;
        void *buffer;
};
```

Each set has the following counters:

```
struct atm_aal_stats {
#define __HANDLE_ITEM(i) int i
        __AAL_STAT_ITEMS
#undef __HANDLE_ITEM
};
```

`tx` and `rx` count the PDUs that have successfully been sent or received, respectively. `tx_err` counts cases where an SDU accepted for transmission was discarded later. `rx_err` counts PDUs received with a bad CRC, an invalid length, or a similar defect. `rx_drop` counts PDUs discarded because of memory shortage.[2] The behaviour on overflows is undefined.

3 VC data

Device-independent VC parameters are stored in `struct atm_vcc`:

```
struct atm_vcc {
        atm_vcc_flags_t flags;
        unsigned char   family;
        short           vpi;
        int             vci;
        unsigned long   aal_options;
        unsigned long   atm_options;
        struct atm_dev  *dev;
        struct atm_qos  qos;
        struct atm_sap  sap;
        atomic_t        tx_inuse,rx_inuse;
        void (*push)(struct atm_vcc *vcc,struct sk_buff *skb);
        void (*pop)(struct atm_vcc *vcc,struct sk_buff *skb);
        struct sk_buff *(*alloc_tx)(struct atm_vcc *vcc,unsigned int size);
        int (*push_oam)(struct atm_vcc *vcc,void *cell);
        int (*send)(struct atm_vcc *vcc,struct sk_buff *skb);
        void            *dev_data;
        void            *proto_data;
        struct timeval  timestamp;
        struct sk_buff_head recvq;
        struct k_atm_aal_stats *stats;
        wait_queue_head_t sleep;
        struct sock     *sk;
        struct atm_vcc  *prev,*next;
        short           itf;
        struct sockaddr_atmsvc local;
        struct sockaddr_atmsvc remote;
        void (*callback)(struct atm_vcc *vcc);
```

[2] Note that `rx_err` and `rx_drop` may not only be incremented by the device driver but also by the `peek` function.

```
        struct sk_buff_head listenq;
        int             backlog_quota;
        int             reply;
        struct atm_vcc  *session;
        void            *user_back;
};
```

`flags` contains flags indicating the VC state. The following flags are relevant for the device driver:

`ATM_VF_ADDR` indicates that the connection identifier stored in `vpi` and `vci` (see below) shall be considered to be "in use". This flag should be set by the device driver immediately before allocating the connection identifier in hardware, and the device driver must clear the flag immediately after freeing the connection identifier in hardware. Note that other entities may set `ATM_VF_ADDR` before invoking the SAR driver's `open` function.

`ATM_VF_READY` indicates that the VC is ready to transfer data. This flag must be set by the device driver when transfers are possible in all requested directions and it should be cleared by the device driver before closing the VC. Note that other entities may clear `ATM_VF_READY` before requesting a close.

`ATM_VF_PARTIAL` indicates that resources are allocated to the (P)VC. This flag is entirely under the control of the device driver.

`family` is the address family, i.e. either `AF_ATMPVC` or `AF_ATMSVC`. `aal` contains the AAL number. `dev` contains a pointer to the corresponding device structure. `txtp` and `rxtp` contain send and receive traffic parameters. All these fields are initialized before the `open` function is called.

`vpi`, and `vci` contain the connection identifier. They have to be set by the `open` function.[3]

`tx_quota`, `rx_quota`, `tx_inuse`, and `rx_inuse` describe the maximum size of the transmit and receive queues (in bytes) and their current utilization. Those quotas are maintained and enforced by the protocol.

`aal_options` and `atm_options` are reserved for future use. The protocol operations `push`, `pop`, `peek`, and `push_oam` are described in section 6. `alloc_tx` can be altered by the protocol, the device driver, or both. It is also described in section 6.

`dev_data` points to device-specific private data, i.e. to a device-specific VC descriptor. Similarly, `proto_data` points to protocol-specific data. Neither of those pointers is ever touched by any other entity than the one owning it.

`stats` points to the corresponding set of counters in the device structure. The device driver shall use `stats` to access statistics counters (as opposed to accessing them via `dev`). `stats` is initialized before `open` is called.

`timestamp`, `recvq`, `sleep`, `prev`, `next`, and all SVC-related fields should not be touched by the device driver.

4 Socket buffer extensions

The socket buffer structure `struct sk_buff` (defined in `skbuff.h`) is extended for ATM with the following fields:

[3] The requested VPI and VCI are passed to `open` as arguments.

`size`, `pos`, and `vcc` are reserved for internal use by the device driver and are not touched by any other entity.

`iovcnt` contains the size of the scatter-gather vector. If no scatter-gather is used, `iovcnt` is set to zero. When using scatter-gather, `skb->data` points to an array of elements of type `struct iovec` and `skb->len` indicates the total length of all buffers. When receiving, `iovcnt` must be initialized by the device driver. When sending, `iovcnt` is appropriately initialized before `send` is invoked.

`timestamp` contains the time at which the cell was received. `timestamp` has to be set by the device driver before calling the `push` function.

5 Device operations

Function pointers for ATM device operations are stored in `struct atmdev_ops`:

```
struct atmdev_ops {
        void (*dev_close)(struct atm_dev *dev);
        int (*open)(struct atm_vcc *vcc,short vpi,int vci);
        void (*close)(struct atm_vcc *vcc);
        int (*ioctl)(struct atm_dev *dev,unsigned int cmd,void *arg);
        int (*getsockopt)(struct atm_vcc *vcc,int level,int optname,
            void *optval,int optlen);
        int (*setsockopt)(struct atm_vcc *vcc,int level,int optname,
            void *optval,int optlen);
        int (*send)(struct atm_vcc *vcc,struct sk_buff *skb);
        int (*sg_send)(struct atm_vcc *vcc,unsigned long start,
            unsigned long size);
#if 0
        int (*send_iovec)(struct atm_vcc *vcc,struct iovec *iov,int size,
            void (*discard)(struct atm_vcc *vcc,void *user),void *user);
#endif
        int (*send_oam)(struct atm_vcc *vcc,void *cell,int flags);
        void (*phy_put)(struct atm_dev *dev,unsigned char value,
            unsigned long addr);
        unsigned char (*phy_get)(struct atm_dev *dev,unsigned long addr);
        void (*feedback)(struct atm_vcc *vcc,struct sk_buff *skb,
            unsigned long start,unsigned long dest,int len);
        int (*change_qos)(struct atm_vcc *vcc,struct atm_qos *qos,int flags);
        void (*free_rx_skb)(struct atm_vcc *vcc, struct sk_buff *skb);
        int (*proc_read)(struct atm_dev *dev,loff_t *pos,char *page);
};
```

`open` is invoked to open a VC in hardware. Many fields in the VC descriptor are already initialized, see section 3. The VPI and the VCI are passed as arguments. Note that they may be wildcarded or incompletely specified. The device driver has to perform the requested resource allocations and possibly select an appropriate VC. `open` returns zero if the operation has succeeded, or a negative error code if the operation has failed. In case of a failure, `open` must undo all state changes (e.g. allocations) before returning.

`close` is invoked to close a VC and to return all resources allocated for it. `close` is only invoked once per VC and only if the `open` operation was successful.

`ioctl` passes a device ioctl to the device driver. Note that for device-private ioctls, `arg` is passed "as is" from the calling process and may point to memory locations that process is not allowed to access.

Memory access and permissions are already checked for "standard" ioctls. If an ioctl write back data to the caller, the function also has to return the length in bytes. If the SAR driver does not recognize an ioctl, it must pass it on to the PHY driver if one is defined and if it provides an `ioctl` function. The list of assigned ioctl values for Linux ATM is available at `http://lrcwww.epfl.ch/linux-atm/magic.html`

`getsockopt` and `setsockopt` can be used to access driver-specific variables. When calling `getsockopt`, `optval` and `optlen` point to unchecked memory. When calling `setsockopt`, read access for `optval` has been checked. Unlike `ioctl`s, unrecognized `getsockopt`s and `setsockopt`s are not forwarded to the PHY driver. Every ATM driver should support at least `SO_CIRANGE`.

`send` is used to enqueue SDUs for transmission. The socket buffer (skb) must remain accessible until the driver explicitly frees it by invoking the `pop` function. If no `pop` function is provided, the driver must call `dev_kfree_skb(skb,FREE_WRITE)` instead. Of the ATM-specific skb fields, only `iovcnt` is initialized when `send` is invoked.

`sg_send` is used by the protocol to determine whether a packet should be sent using scatter-gather. Drivers that don't implement scatter-gather should set `sg_send` to NULL. `sg_send` examines the start address and the length of the packet to estimate whether there is a performance advantage in locking the packet in memory, building a scatter-gather vector, and sending directly from user space (see also [1]), as opposed to copying the data to a kernel buffer and sending that buffer. `sg_send` returns zero if scatter-gather should not be attempted, a non-zero value otherwise.

`poll` is invoked before the protocol checks for the presence of received data. Drivers which are not notified (e.g. by interrupts) when new packets arrive need this function to poll their respective data source. Interrupt-driven drivers should set `poll` to NULL. The `nonblock` argument indicates whether the `poll` function should wait until something has been received (if zero) or not (if non-zero). `poll` may be removed at a later time (a driver could also use timer-driven polling with similar or better results).

`send_oam` is used to send OAM cells. `cell` points to the cell in kernel memory. The same format as for AAL0 cells is used (see [2]). `flags` indicates how the cell should be transmitted. The following flags are defined:

`ATM_OF_IMMED` immediate delivery requested (instead of in-sequence)

`ATM_OF_INRATE` in-rate delivery requested

If a driver is unable to support the requested transmission, it should not fail the operation but it should use the closest available transmission method (e.g. send immediately even if `ATM_OF_IMMED` is not set) instead. `send_oam` may block for a short moment when performing in-sequence or in-rate emission. Note that `send_oam` may be invoked from an interrupt and must therefore not try to sleep. `send_oam` is currently not used.

`phy_put` is invoked by the PHY driver to write one byte to a register of the PHY device. `phy_get` is used to read a byte. Both functions may be invoked from interrupts. A SAR driver must provide those two functions if it uses a PHY driver (and if the PHY device supports the corresponding operation).

`feedback` may be invoked when the protocol or the socket has determined at which memory address an incoming SDU will be stored. The start address of the application-level SDU (i.e. after removing headers and trailers), the destination address, and the size of the application-level SDU are passed to `feedback`. This information can be used to adjust parameters of heuristics used by the driver to predict optimal buffer allocation for receiving. Note that the protocol is not obliged to ever call `feedback` upon delivery of a packet.

In addition to the functions in `struct atmdev_ops`, each SAR driver must also provide a device detection function which is invoked by `atmdev_init` (in `atmdev_init.c`).

6 Protocol operations

Protocol operations invoked by the device driver are defined in `struct atm_vcc`:

```
void (*push)(struct atm_vcc *vcc,struct sk_buff *skb);
void (*pop)(struct atm_vcc *vcc,struct sk_buff *skb);
struct sk_buff *(*alloc_tx)(struct atm_vcc *vcc,unsigned int size);
int (*push_oam)(struct atm_vcc *vcc,void *cell);
```

`push` is invoked by the ATM driver to deliver received packets to the protocol. Every protocol must provide a `push` function. The protocol acquires ownership of the skb and is responsible for freeing it (with `kfree_skb(skb,FREE_READ)`). `push` is invoked by the socket layer with the `skb` argument set to NULL when the VC is closed, in order to detach the protocol from the VC. Note that the device driver `close` function is called before detaching the protocol, so that the VC can no longer be used to send or receive data.

`pop` is invoked after successful emission of a SDU to indicate that the ATM driver will no longer use the skb. If no `pop` function is provided (i.e. if `pop` is set to NULL), the driver calls `dev_kfree_skb(skb,FREE_WRITE)` instead.

`peek` is invoked by a device driver whenever a new buffer has to be allocated for an incoming packet. The size of the packet and, optionally, a pointer to a function to examine the new packet are passed to the protocol. The `fetch` function can be used to read the `i`th 32 bits word from the packet, e.g. in order to improve buffer alignment based on protocol or service information contained in a packet header. Drivers which cannot provide such functionality must pass NULL instead. `peek` either returns an skb large enough for the packet or it returns NULL, indicating that the packet should be discarded. `peek` has to update the statistics (`vcc->stats`) in this case to indicate whether the packet was discarded due to lack of memory (`rx_drop`) or due to some other problem (`rx_err`).

`alloc_tx` is used to allocate kernel memory for sending a datagram. Before protocol initialization and invocation of the device driver `open` function, `alloc_tx` is set to a function that only invokes `alloc_skb`. Both, the protocol and the device driver can specify their own functions instead. A replacement function should always invoke the function previously found in `alloc_tx` to perform the actual allocation. This way, additional memory or alignment requirements of drivers and protocols can easily be combined.[4]

If non-NULL, `push_oam` is invoked whenever an OAM cell is received. A pointer to the cell content is passed. The cell is stored in the same format as used for AAL0. `push_oam` can request in-sequence delivery of the cell by returning a non-zero value. If `push_oam` returns zero, the cell is discarded afterwards. If `push_oam` is set to NULL, all OAM cells are delivered in-sequence. Hardware limitations may only allow approximative in-sequence delivery.

7 Physical layer device operations

Function pointers for physical layer device driver operations are stored in `struct atmphy_ops`:

```
struct atmphy_ops {
        int (*start)(struct atm_dev *dev);
        int (*ioctl)(struct atm_dev *dev,unsigned int cmd,void *arg);
        void (*interrupt)(struct atm_dev *dev);
        int (*stop)(struct atm_dev *dev);
};
```

[4] The arguments of `alloc_tx` will be changed in future releases to support scatter-gather.

`start` is invoked by the SAR driver to start the PHY device. This function is called at least once per interface.

`ioctl` is invoked by the SAR driver to forward an ioctl that is not understood by it. Each PHY driver for SONET should at least support the `SONET_GETSTATZ`, `SONET_GETSTAT`, `SONET_SETFRAMING`, and `SONET_GETFRAMING` ioctls. In addition to that, the ioctls `SONET_SETDIAG`, `SONET_CLRDIAG`, and `SONET_GETDIAG` should be supported if the hardware has the capability to simulate errors. PHY drivers that don't support `ioctl` should set the field to NULL.

`interrupt` is invoked whenever the SAR driver receives an interrupt which is triggered by the PHY device. This function is only necessary if PHY interrupts are relayed through the SAR device and if the PHY generates interrupts at all.

In addition to the functions in `struct atmphy_ops`, each PHY driver must also provide an initialization function which is invoked by the SAR driver. That function should initialize the PHY device and it has to populate the `dev->phy` field.

8 Support functions

Some device-independent support functions are defined in `atmdev.h`:

```
struct atm_dev *atm_dev_register(const char *type,const struct atmdev_ops *ops,
    int number,atm_dev_flags_t *flags);
struct atm_dev *atm_find_dev(int number);
void atm_dev_deregister(struct atm_dev *dev);
void shutdown_atm_dev(struct atm_dev *dev);
void bind_vcc(struct atm_vcc *vcc,struct atm_dev *dev);
 * This is approximately the algorithm used by alloc_skb.
 *
 */
static __inline__ int atm_guess_pdu2truesize(int pdu_size)
{
        return ((pdu_size+15) & ~15) + sizeof(struct sk_buff);
}
static __inline__ void atm_force_charge(struct atm_vcc *vcc,int truesize)
{
        atomic_add(truesize+ATM_PDU_OVHD,&vcc->rx_inuse);
}
static __inline__ void atm_return(struct atm_vcc *vcc,int truesize)
{
        atomic_sub(truesize+ATM_PDU_OVHD,&vcc->rx_inuse);
}
static __inline__ int atm_may_send(struct atm_vcc *vcc,unsigned int size)
{
        return size+atomic_read(&vcc->tx_inuse)+ATM_PDU_OVHD < vcc->sk->sndbuf;
}
int atm_charge(struct atm_vcc *vcc,int truesize);
struct sk_buff *atm_alloc_charge(struct atm_vcc *vcc,int pdu_size,
    int gfp_flags);
int atm_find_ci(struct atm_vcc *vcc,short *vpi,int *vci);
```

`atm_dev_register` registers a new ATM device. Pointers to the interface type name and to the device driver operations structure have to be passed. Note that the type name must be accessible at the

specified address until after the device is de-registered. `atm_dev_register` either returns a pointer to a new, initialized device structure, or NULL if no device structure was available for allocation.

`atm_dev_deregister` de-registers an ATM device. All activity on that device must have been stopped before invoking `atm_dev_deregister`. `atm_dev_deregister` is typically used to deallocate device structures after an initialization failures.

`atm_find_ci` determines whether the specified connection identifier is available on `vcc->dev`. If a wildcard is specified, `atm_find_ci` tries to locate an unused connection identifier. Note that `atm_find_ci` is comparably inefficient and that a device driver may be able to perform noticeably faster lookups on its internal tables. On success, `atm_find_ci` sets `vcc->vpi` and `vcc->vci` and returns zero. Otherwise, it returns a negative error code.

9 Summary

Table 1 summarizes ownership of fields contained in ATM-specific data structures. Only fields relevant to ATM device drivers are shown. Most of the fields are read by various system components, so cross-references for read accesses would be of only marginal use.

Structure	Field	Initialized by	Modified by
atm_dev	ops	atm_dev_register	—
atm_dev	phy	PHY driver initialization (if applicable)	—
atm_dev	type	atm_dev_register	—
atm_dev	dev_data	SAR driver (if using field)	SAR driver
atm_dev	phy_data	PHY driver (if using field)	PHY driver
atm_dev	esi	atm_dev_register, SAR driver initialization	SAR driver
atm_dev	ci_range	SAR driver initialization	SAR driver
atm_dev	stats	atm_dev_register[5]	SAR driver, protocol
atm_vcc	flags	ATM socket	Many components
atm_vcc	aal	ATM socket, protocol	—
atm_vcc	dev	ATM socket	ATM socket[6]
atm_vcc	txtp	ATM socket	SAR driver
atm_vcc	rxtp	ATM socket	SAR driver
atm_vcc	vpi	SAR driver	—
atm_vcc	vci	SAR driver	—
atm_vcc	aal_options	ATM socket	ATM socket
atm_vcc	atm_options	ATM socket	ATM socket
atm_vcc	proto ops	Protocol	Protocol[7]
atm_vcc	alloc_tx	ATM socket	SAR driver, protocol
atm_vcc	dev_data	SAR driver	SAR driver
atm_vcc	proto_data	Protocol	Protocol
atm_vcc	stats	ATM socket, protocol	—
sk_buff	atm.size	SAR driver (if using field)	SAR driver
sk_buff	atm.pos	SAR driver (if using field)	SAR driver
sk_buff	atm.vcc	SAR driver (if using field)	SAR driver
sk_buff	atm.iovcnt	TX: ATM socket, protocol	—
		RX: SAR driver	—
sk_buff	atm.timestamp	SAR driver	—

Table 1: Summary of field ownership

Table 2 summarizes the operations that are provided by ATM device drivers (SAR and PHY part) and by ATM protocol modules. Again, only fields relevant to ATM device drivers are shown. "interrupt" indicates that a function might be invoked during an interrupt.

Structure	Field	Optional	Provider	Caller
atmdev_ops	open	Yes	ATM driver	Protocol
atmdev_ops	close	Yes	ATM driver	Protocol
atmdev_ops	ioctl	Yes	ATM driver	ATM socket
atmdev_ops	getsockopt	Yes	ATM driver	ATM socket
atmdev_ops	setsockopt	Yes	ATM driver	ATM socket
atmdev_ops	send	No	ATM driver	Protocol
atmdev_ops	sg_send	Yes	ATM driver	Protocol
atmdev_ops	poll	Yes	ATM driver	Any non-interrupt
atmdev_ops	send_oam	Yes	ATM driver	Protocol, interrupt
atmdev_ops	phy_put	No[8]	SAR driver	PHY driver, interrupt
atmdev_ops	phy_get	No[8]	SAR driver	PHY driver, interrupt
atmdev_ops	feedback	Yes	ATM driver	ATM socket
atm_vcc	push	No	Protocol	ATM driver, interrupt
atm_vcc	pop	Yes	Protocol	ATM driver, interrupt
atm_vcc	peek	No	Protocol	ATM driver, interrupt
atm_vcc	alloc_tx	No	ATM drv., proto.	ATM socket
atm_vcc	push_oam	No	Protocol	ATM driver, interrupt
—	fetch	Yes	ATM driver	Protocol, interrupt
—	Detect	No	ATM driver	atmdev_init
atmphy_ops	start	No	PHY driver	SAR driver
atmphy_ops	ioctl	Yes	PHY driver	SAR driver (called by socket)
atmphy_ops	interrupt	No[9]	PHY driver	SAR driver, interrupt
—	Initialization	No	PHY driver	SAR driver
—	atm_dev_register	No	Device-indep.	ATM driver
—	atm_dev_deregister	No	Device-indep.	ATM driver
—	atm_find_ci	No	Device-indep.	ATM driver

Table 2: Summary of protocol and driver operations

References

[1] Almesberger, Werner. *High-speed ATM networking on low-end computer systems*, ftp://lrcftp.epfl.ch/pub/linux/atm/papers/atm_on_lowend.ps.gz, August 1995.

[2] Almesberger, Werner. *Linux ATM API*, ftp://lrcftp.epfl.ch/pub/linux/atm/api/, February 1996.

[5] Zeroed by device-independent functions.

[6] This modification occurs before the ATM device driver gets involved.

[7] E.g. IP over ATM redirects the **push** function if the ATMARP demon requests to use a VC for IP traffic.

[8] A SAR driver may not provide **phy_put** or **get_put** functions if all the PHY drivers it uses are known to test for unavailability of those functions.

[9] Only if interrupts are generated by the physical layer device and if they are relayed through the SAR driver.