ISDN4Linux, CAPI4Linux, CAPI4Hisax and other cute acronyms: The ISDN subsystem in the Linux kernel

Kai Germaschewski
University of Iowa, Department of Physics and Astronomy
203 Van Allen Hall, Iowa City, IA 52242, USA

(Dated: August 14, 2002)

During the Linux kernel 2.5 development cycle, the ISDN subsystem in the Linux kernel will undergo major changes. This paper gives an outline of the layers of the legacy ISDN code, called ISDN4Linux. The core part of that code, called the ISDN link layer, is a multi/demultiplexing layer which coordinates data exchange between hardware drivers on one side and applications using ISDN service on the other side.

We will show how replacing this layer with a CAPI based solution, called CAPI4Linux, benefits both hardware drivers and applications. CAPI is an open standard ("Common ISDN Application Programming Interface"), which provides a standard interface to using ISDN communication services independently of the actual hardware and drivers used. In contrast to the old ad-hoc interface which was extended bit by bit to match growing requirements, CAPI is a well known and documented interface. Porting userspace applications from and to other OS's is simplified a lot using a common API. On the driver side, an obvious advantage is accomplished for so-called active ISDN cards, which implement the CAPI interface in firmware. They can now be interfaced directly to the kernel CAPI layer without being translated back and forth from the old ISDN link layer API.

However, an overwhelming market share is owned by so-called passive ISDN adapters. They do not come with a processor of their own but expect the host processor to handle the ISDN protocols, call control and the like. Linux has a driver which handles almost all existing passive cards and implements the necessary protocol layers. Currently this driver ("hisax") interfaces to the old ISDN4Linux API, rendering it unusable with the CAPI4Linux subsystem. We describe the conversion of this driver to an open source CAPI compliant driver (the project was named "CAPI4HiSax"), which is an essential building block on the way towards the new CAPI based ISDN layer.

INTRODUCTION

ISDN (Integrated Services Digital Network) was introduced as the successor to the so-called POTS (plain old telephony system). In course of the digital age, telephone companies replaced old analogue or even mechanical switches with modern digital exchanges. Audio is digitized right at the interface to the local exchange, and all further processing happens digitally, until the digital stream is converted back to the audio at the interface to the receiver's phone line.

Digital exchanges provide a large set of additional features, starting from services like caller ID and extending to call forwarding and conferencing. The natural way to provide access to these features is to extend the digital part of the connection into the customer's home, and this is what ISDN does.

A normal basic rate ISDN interface provides two B-channels and one D-channel on the so-called digital S0 bus installed in the customer's home. A B-channel offers 64 kbit/s bandwidth, which can be used to transfer speech in normal phone quality or to transfer digital data directly without first converting it to audio using a modem (which would than get digitized again at the local exchange before further transmission). Call control and additional services are handled out-of-band using the D-channel which has 16 kbit/s bandwidth. So the number we want to call is not transmitted on the actual line using DTMF encoding, but by sending a digital SETUP message on the D-channel. The exchange will assign a B-channel to the connection when it gets established.

Setting up a digital connection works the same way, sending a SETUP message on the D-channel, this time indicating that the B-channel will be used for *unrestricted digital information*. The entire connection setup can happen in less than one second as opposed to about 30 seconds for a modem connection. For one reason, sending the number directly in a digital message is faster than converting it to DTMF tones and transferring those. The main advantage is that the connection is entirely digital end-to-end, though, i.e. the modulation / demodulation to audio and the related handshaking and modem training are not necessary.

Computers can be hooked up to the ISDN S0 bus directly, using ISDN adapters, which may be actual ISA or PCI extension boards or external devices which connect via USB, parallel port or a serial port. Some devices actually present themselves like a modem to the computer. These devices can be treated exactly like an analogue modem from the computer's point of view and are thus not covered in this paper, which deals with devices which present the actual ISDN interface to the computer.

This paper is structured as follows: We start with giving an overview of the history of the ISDN subsystem in the Linux kernel. Detailing the evolution of ISDN4Linux, it also becomes clear which deficiencies exist in the current solution. We will show how these deficiences can be overcome during the move to a new CAPI based ISDN subsystem. First, we introduce the general concept of CAPI, which is an hardware and operating system independent programming interface for using ISDN services. We then show how this API has been implemented in the Linux kernel and at the userspace level.

Finally, we present an overview of the work which needs to be done to adapt the HiSax driver to the new CAPI based subsystem. It consists of splitting HiSax into the protocol handling part and actual hardware drivers on the other hand. Furthermore, the interface to the old ISDN4Linux link layer will be replaced by an interface to the kernel CAPI layer.

HISTORY OF THE ISDN SUBSYSTEM IN THE LINUX KERNEL

In the beginning of the ISDN days, two packages to support ISDN cards under Linux were developed, the *Urlichs ISDN* by Matthias Urlichs and *ISDN4Linux* by Fritz Elfert. This was roughly in the Linux kernel 1.2 days, and since at that time the author did not care about ISDN in the least, he cannot really comment on specific advantages or disadvantages. At a late point during the 1.3 Linux kernel development cycle, version 1.3.69, ISDN4Linux was merged into Linus Torvalds' official kernel tree.

Beginning at that time, ISDN4Linux was officially part of the Linux kernel which has its advantages, for example easy availability to interested people who do not want to go through the trouble of downloading CVS developer versions and patching their kernels, but it also brought downsides, in particular ISDN became Linus Torvalds' most prominent example on how *not* to do development, maintainance and merging with him.

The author first came into contact with ISDN as he moved in together with a roommate and they decided to get an ISDN line. Planning to use ISDN under Linux, he purchased a Elsa Quickstep 1000 PCI ISDN card, since Elsa supplied the Linux ISDN developers with hardware specs and also supported the certification of the HiSax driver under the ITU approval tests.

These were about kernel 2.0.35 times, the ISDN4Linux in the kernel was up-to-date and in good shape. The original teles driver by Jan van den Ouden had just been replaced by the HiSax driver written by Karsten Keil, which supported a number of additional passive ISDN cards. However, the Redhat distribution, at that time mainly targeted on the Northamerican market, did not support ISDN out of the box, so setting up ISDN services needed some handmade scripting, which however had the nice side effect of getting familiar with the concepts and commands of ISDN4Linux.

The main focus of ISDN4Linux was supporting network device and emulated modem support on top of ISDN, particularly useful for setting up connections between remote computers and also providing some answering machine type services analogous to those of voice capable modems.

The author's first real involvement with the kernel ISDN stack happened when he decided to write some tool to activate and deactivate Call Forwarding services using the ISDN card. Here, open source really showed its strengths, having the sources available and with the necessary standards documents from the ITU/ETSI at hand, it provides the possibility to everyone to adapt and extend to their needs. The main problem in the implementation was actually to find a clean way to interface to userspace, since ISDN4Linux was not designed to handle supplementary services like call forwarding.

In the 2.1 development cycle, the ISDN maintainance problems already known from 1.3 (inclusion after code freeze) and 2.0 recurred. The ISDN developers preferred working on their CVS development tree and only after they were sufficiently happy with a development version which was stable and tested, they would try to sync with Linus Torvalds.

This way of maintainance is very much incompatible with Linus Torvalds, who prefers to get frequent nicely separated patches which fix one problem at a time instead of large patches once in a while. The latter kind tends to basically replace one version of the subsystem with a new one, containing a large mix of fixes, cleanups and improvements, making it impossible to grasp what each change does. So both sides got frustrated with the not working merging process during 2.1 and the development version diverged even more from the kernel code. Since the version in the Linux development tree was basically unmaintained, it became unusable, so people using or testing ISDN had to use the CVS development version, adding to the deserted state of ISDN4Linux in the kernel tree. Again, as code freeze time approached for 2.1, the ISDN developers realized that it was time to push their new version to Linus, but due to the large amount of changes and late time in the development cycle, Linus rejected the patches and Linux kernel 2.2 appeared with a mostly unusable ISDN4Linux subsystem. Users were forced to fall back to the ISDN4Linux CVS version, and with noone actually using the broken version in the official 2.2, it took a long time to have it fixed and updated. Eventually, with the help of Alan Cox, who had a more pragmatic approach to maintainance, kernel versions after 2.2.12 or so had a usable ISDN subsystem.

Another issue had contributed to the ISDN4Linux maintainance problems, the fact that a lot of work was spent on ISDN4Linux at about 2.0.30, adding substantial improvements and a new driver, HiSax, which supported many new passive cards. Since 2.0 and 2.1 development was done on two different CVS branches, these branches diverged and 2.0 was quickly far ahead of the 2.1, causing a lot of effort to get the versions resynced again later.

To avoid this kind of problem, it was decided to keep only one CVS branch from that time on, which would work for all supported kernel versions. Karsten Keil created a small parser, which would generate code for 2.2 and 2.3/2.4 out of that common source.

This approach worked nicely for many small trivial changes (PCI resource handling, new wait queue interface etc.), and it also worked sufficiently well for some larger adaptions, as for example the updated softnet/tbusy handling. On the other hand, it was not able to support larger changes, like for example a general cleanup of the ISDN4Linux link layer, replacing the global cli()/sti() locking with spinlocks and similar things. Such changes, while appropriate for 2.3, would have been too large and experimental for 2.2, so core changes were kept to a minimum allowing for a working ISDN4Linux in kernels 2.2 and 2.3/4. Another factor was of course that a working ISDN layer gave little incentive to spend a lot of time rewriting things (and thereby breaking it in all possible ways at least once), so most work was spent on additional hardware support, mostly by Karsten Keil, Werner Cornelius and Armin Schindler.

The author, initially motivated by the lack of a sensible userspace for extensions of ISDN4Linux as the Call Forwarding services, spent a lot of time working on a CAPI interface for HiSax, and other additional features like an ASN.1 parser needed for the protocol side of several supplementary services. Doing this development outside of the normal CVS repository and of course outside of the development kernel, he would learn a lot of valuable lessons about diverging versions and merge problems which lead to a point where it actually makes sense to start over rather then attempting an impossible merge.

At the same time, the 2.3 development cycle was getting closer to the stabilizing phase, and again the ISDN4Linux version in the development kernel was getting neglected. At this point, the author started to make necessary adaptions himself, and commit them to the ISDN4Linux repository. After some time of seeing these nicely separated changes accumulate to big patches between CVS and Linus' version, he decided to try to submit patches to Linus himself, reducing the diff size between 2.3 and CVS. Most of the ISDN developers preferred to spend their time doing actual coding and testing instead of doing the tedious chore of separating and explaining patches, and emailing them to Linus. So the author took over this task, learning gradually about the way Linus prefers his patches, when to resubmit patches when they have been silently dropped, and of course about the excitement of seeing patches one submitted or even wrote oneself getting merged into Linus' official tree. After not too long, the merging process worked pretty well, and eventually, after having performed the task for about a year, the author decided to add himself to the MAINTAINERS file in 2.4.4.

Not too much happened during the 2.4 kernel series, some preparational went in for breaking up the monolithic HiSax design, which traditionally is one big module which supports more than 40 different kinds of ISDN hardware. The goal is to have HiSax as the protocol layer in one module, and then add additional modules for the different kind of ISDN boards, also catering for hotpluggable hardware.

Now, going from the past to the future, what is expected to be done during the 2.5 development cycle? The stated goal is to declare the old ISDN4Linux link layer obsolete and replace it by a new CAPI-centric subsystem. The active AVM cards have always supported CAPI directly, using their onboard processors and firmware, and a CAPI subsystem with the basic features (CAPI interface to userspace and PPP connectivity) exists. 2.5 has seen some restructuring of the ISDN code layout and improvements and cleanups to the CAPI framework, though that is not entirely finished yet. So the most needed step is to convert the HiSax driver to the new CAPI interface – the new code exists, but it will be a substantial amount of work to merge it to current 2.5 and break it up into usable pieces for submission to Linus. So the ISDN subsystem is on track, but lagging behind the timeline the author would have imagined. Since the framework part of the work is accepted and part of the tree, the remaining work is not adding features, but rather cleaning up and adapting, so even if part of that work happens after feature freeze, the stated goal seems achievable until 2.6 is released.

ISDN4LINUX

Using material from an ISDN workshop in 1998 [6], we outline the basic structure of ISDN4Linux in the following. Nothing substantial did change since that time, though new features were added, some times further obfuscating the code.

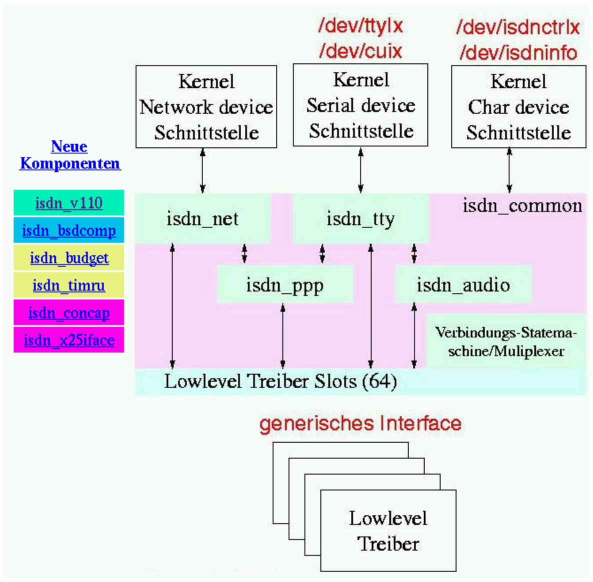


FIG. 1: Basic structure of ISDN4Linux

ISDN4Linux has two major components: Card drivers and the ISDN link layer (Figure 1. The link layer serves two purposes: On the hand it provides a hardware independent interface to hardware drivers for a variety of ISDN cards, on the other hand it interfaces to standard kernel subsystems, providing network interfaces and tty's.

ISDN4Linux has provided users with the ISDN they needed over a wide range of Linux kernel releases. However, unlike most other subsystems, it has never undergone a major cleanup or rewrite. The consequence is that deficiencies have become visible which we list in the following to show how the new CAPI based solution will improve the situation.

One major drawback is that only the interface between link layer and drivers has been abstracted, no interface between the in-kernel applications and a common layer providing connection setup and similar services has been created. So status and even data messages from the hardware drivers are just passed to the different submodules one after another until one finally accepts them:

```
if (isdn_net_stat_callback(i, c))
        return 0;
if (isdn_v110_stat_callback(i, c))
        return 0;
if (isdn_tty_stat_callback(i, c))
        return 0;
wake_up_interruptible(&dev->drv[di]
        ->snd_waitq[c->arg]);
```

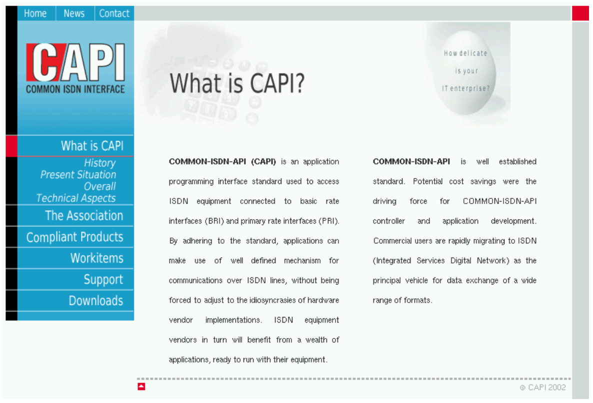
This also leads to unnecessary code duplication in `isdn_tty.c` and `isdn_net.c`, since both have to handle connection setup and teardown by themselves.

The interface between hardware drivers and link layer is not specified via an exact state machine, nor is the context in which the callbacks happen defined, leading to subtle differences between different drivers and the potential of the code magically breaking under certain circumstances that are difficult to debug.

Additionally, not all indications available from the ISDN network are promoted to the ISDN link layer, rendering certain actions impossible. For example, no indication is available for alerting, which is a message sent by the called device indicating that it started ringing or something equivalent. So to initiate a callback connection, we cannot just dial the number, wait until we get the alerting indication, and hang up, knowing that the call reached the other side. Instead we just wait for a certain amount of time after setup, hoping to guess the right delay, long enough for the other side to notice our call, and short enough to not be still busy when the callback comes in.

The driver / link layer interface is based on the concept of channels, which correspond to B-channels on the hardware. However, that does not offer sufficient flexibility for all applications. For example, it is possible that an application only wants to use some supplementary services (e.g. setting up call forwarding), which actually does not need any B-channel access at all, and one does not want to block a channel on the interface for such an application. On the other hand, it is perfectly possible to have more than <*number of B-channels*> calls established at the same time, using features like call waiting, hold/retrieve or three-way conference calling.

One last issue with ISDN4Linux is that it actually tries to do too much in kernelspace, which would much better be done in userspace. When using one's modem to dial-up to some PPP provider, pppd/chat will take care of the dialing process from userspace. Only after the connection is established the connection will be switched to do transmiting/receiving PPP data packets inside the kernel, showing up as a pppX network interface. ISDN4Linux, on the other hand, does the entire connection setup and teardown inside the kernel, only using a userspace ipppd to do the actual PPP negotation. Apart from the added complexity inside kernelspace, it also proves rather inflexible. The kernel needs to know about the numbers to dial out to, which numbers to allow dial in from, handle callback in and out, after which period of inactivity to hang up and much more. ISDN4Linux handles the mentioned issues, but there remain always requests for additional features, like e.g. triggering a dial-in only for certain kind of IP packets. Complex code in the kernel is only justified, when similar action cannot be achieved from userspace or is ruled out by performance reasons. The decision when and how to dial up to an ISP is certainly not



What is CAPI?

COMMON-ISDN-API (CAPI) is an application programming interface standard used to access ISDN equipment connected to basic rate interfaces (BRI) and primary rate interfaces (PRI). By adhering to the standard, applications can make use of well defined mechanism for communications over ISDN lines, without being forced to adjust to the idiosyncrasies of hardware vendor implementations. ISDN equipment vendors in turn will benefit from a wealth of applications, ready to run with their equipment.

COMMON-ISDN-API is well established standard. Potential cost savings were the driving force for COMMON-ISDN-API controller and application development. Commercial users are rapidly migrating to ISDN (Integrated Services Digital Network) as the principal vehicle for data exchange of a wide range of formats.

FIG. 2: What is CAPI? - Introduction on www.capi.org

performance-critical, though, and can thus much better be handled from userspace.

COMMON-ISDN-API: CAPI

The established standard programming interface for using ISDN services is the so-called *Common ISDN Application Programming Interface*: CAPI. It provides a hardware and call control protocol independent way of accessing virtually all available ISDN services from applications. A good introduction is given on www.capi.org [3] (see Figure 2).

In the following, a short introduction to the usage of CAPI will be given, presenting some typical examples of message flow. This is not intended to be a course in CAPI programming, though. The complete standards are available for free download on the internet [4].

The first step in using CAPI services is registering the application. using a call to `capi20_register()`.

```
unsigned
capi20_register(unsigned MaxLogicalConnection,
                unsigned MaxBDataBlocks,
                unsigned MaxBDataLen,
                unsigned *ApplIDp);
```

Apart from setting some basic parameters, this call returns an application id (ApplId) which needs to be used in calls to all other CAPI functions. As every CAPI function, the `capi20_register` returns an error code or success, respectively.

When finished using CAPI services, an application should unregister using `capi20_release()`.

```
unsigned
capi20_release(unsigned ApplID);
```

Most of the functionality is accessed by exchanging messages between the application and the CAPI, using `capi20_put_message()` and `capi20_get_message()`.

```
unsigned
capi20_put_message(unsigned ApplID,
                   unsigned char *Msg);

unsigned
capi20_get_message(unsigned ApplID,
                   unsigned char **Buf);
```

This API is operating system independent, so it is possible to share source code between for example Windows 98/2000/.. and Linux. Internally, on Linux `libcapi20.so` basically maps these operations onto a char device interface to the kernel, with the correspondence

```
capi20_register()    -> open()
capi20_release()     -> close()
capi20_put_message() -> write()
capi20_get_message() -> read()
```

Using `capi20_fileno()` it is possible to obtain the file descriptor for a given application id, enabling it to be used for example in normal `select()` loops, at the expense of sacrificing compatibility to operations systems other than Linux.

Messages are always acknowledged by the other side. An application normally sends requests (`*_REQ`) messages to the CAPI, which are confirmed by a corresponding confirmation (`*_CONF`) message from the CAPI. Messages sent from the CAPI to the user are indications (`*_IND`) and need to be answered with a response (`*_RESP`).

Messages are exchanged between the application and specific entities. The message contains a target address, which can either be a controller (physical ISDN interface), a Physical Link Connection Identifier (PLCI), which corresponds to a call/connection on that interface or a Network Control Connection Identifier (NCCI), which is a logical connection on top of the physical one. (Some protocols allow for multiple logical connections (NCCIs) on top of one physical connection (B-Channel), this feature is rarely used, though.)

State machine diagrams are provided that exactly describe which state an entity is in, and which messages lead to state transitions (Figures 3 and 4).

To show that the control flow in practice is much less complicated than what the above may imply for people not familiar with state machines, Figure 5 shows the message flow for a normal dial-out connection.

CAPI4LINUX

The central piece of CAPI4Linux is actually rather simple: It is the module providing registration services to CAPI drivers and in-kernel CAPI applications.

For an active AVM card, the driver is comparatively simple, it basically takes only care of ex-

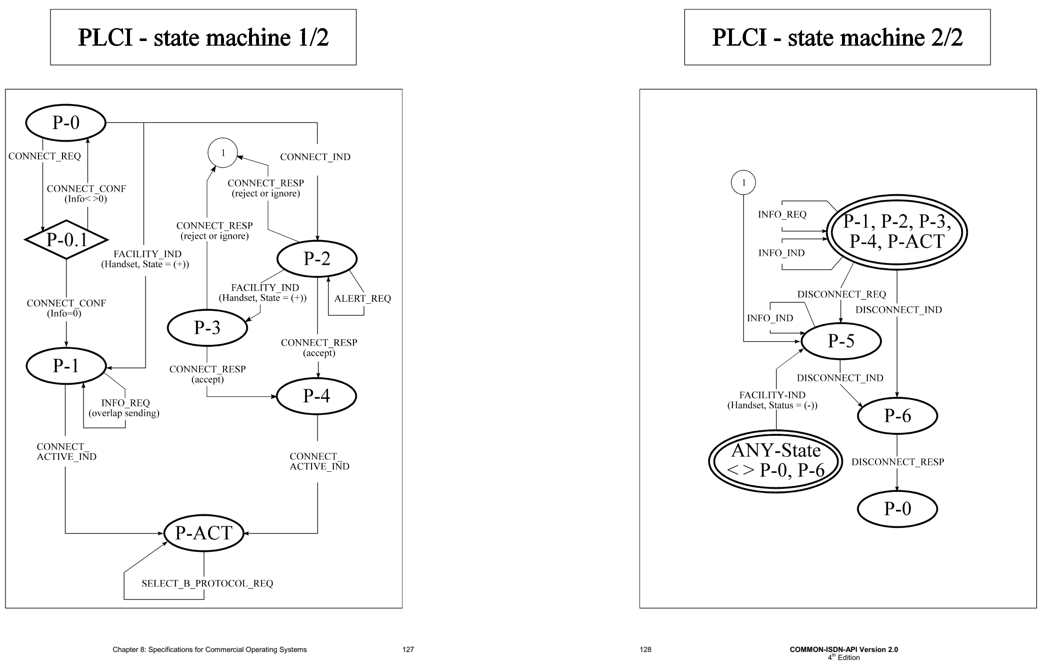


FIG. 3: The PLCI state machine, from [4]

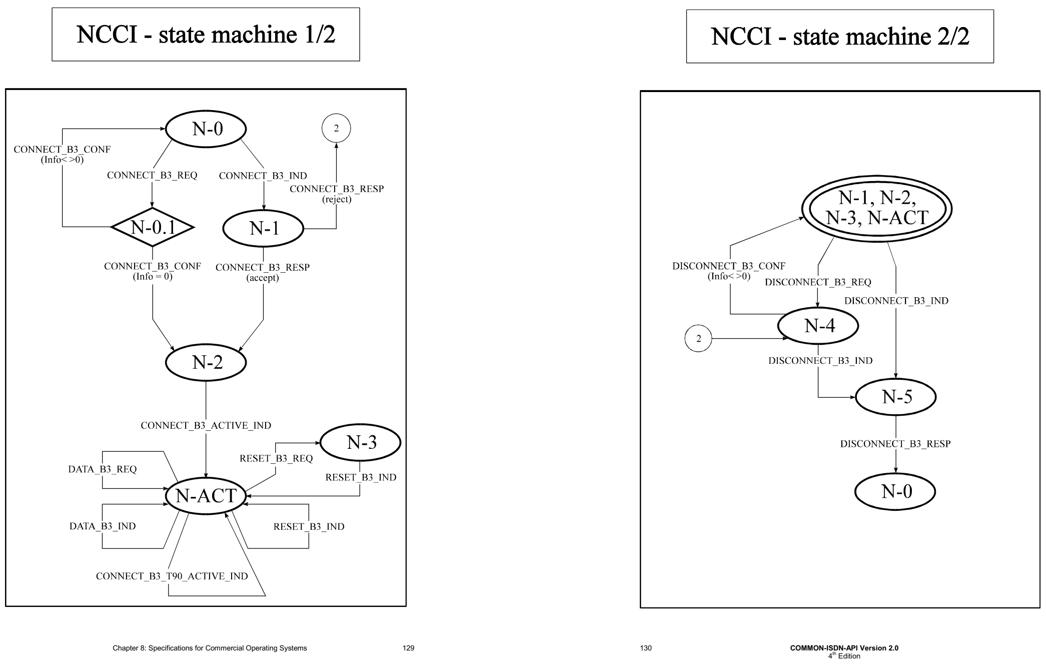


FIG. 4: The NCCI state machine, from [4]

changing messages between kernelcapi and the onboard processor.

For a passive ISDN card, a CAPI driver is much more elaborate, it needs to provide ISDN layer 1 to layer 3 services for both D- and B-channels, as well as driving the actual hardware. On top of this, it has to convert CAPI messages to and from the upmost layer, so CAPI can be considered a layer 4 on top of the traditional ISDN layers 1 to 3. As HiSax already supports all layer 1 to 3 services, the main work consists in adding the needed conversion layer.

The most important in-kernel CAPI application is actually a forwarding module to userspace, providing CAPI services to userspace applications via a `libcapi20.so`. The user space library interface

```
--> CONNECT_REQ(.Controller = 0x01,
                .CIPValue = 1,   /* digital */
                .CallingPartyNumber = 5551234,
                .CalledPartyNumber  = 022156789,
                .B1Protocol = 0, /* HDLC */
                .B2Protocol = 0, /* transparent */
                .B3Protocol = 0) /* transparent */

<-- CONNECT_CONF(.PLCI = 0x101,
                  .Info = 0)      /* initiated */


<-- CONNECT_ACTIVE_IND(.PLCI = 0x101)

--> CONNECT_ACTIVE_RESP(.PLCI = 0x101)


--> CONNECT_B3_REQ(.PLCI = 0x101)

<-- CONNECT_B3_CONF(.NCCI = 0x10101,
                     .Info = 0)    /* initiated */


<-- CONNECT_B3_ACTIVE_IND(.NCCI = 0x10101)

--> CONNECT_B3_ACTIVE_RESP(.NCCI = 0x10101)


[Actual data exchange using DATA_B3_REQ /
 DATA_B3_IND messages]


--> DISCONNECT_B3_REQ(.NCCI = 0x10101)

<-- DISCONNECT_B3_CONF(.NCCI = 0x10101,
                        .Info = 0) /* initiated */


<-- DISCONNECT_B3_IND(.NCCI = 0x10101)

--> DISCONNECT_B3_RESP(.NCCI = 0x10101)


--> DISCONNECT_REQ(.PLCI = 0x101)

<-- DISCONNECT_CONF(.PLCI = 0x101,
                     .Info = 0)    /* initiated */


<-- DISCONNECT_IND(.PLCI = 0x101)

--> DISCONNECT_RESP(.PLCI = 0x101)
```

FIG. 5: Calling out using CAPI

is to remain fixed, providing the same services to an application as on various other operating systems, most notably Windows. The interface between library and kernel is currently provided by a single character device, though this may possibly be changed at a later time.

The main building block of CAPI4Linux is the module `kernelcapi.o`. Apart from providing various library-type utility functions, its main purpose is to take care of the message exchange between applications and drivers. Messages from an application are adressed to a specific controller, so `kernelcapi.o` will queue the message to the driver for the addressed controller. In the other direction, messages from a controller are targeted to a given application, so again `kernelcapi.o` will take care of forwarding it there. To be able to serve these purposes, it needs to keep track of applications and drivers, which it can easily do, as it provides the registration/unregistration interface for both CAPI drivers and in-kernel applications.

As an example, we show how the driver for the AVM B1 PCI card registers a CAPI controller.

```
cinfo->ctrl.driver_name   = "b1pci";
cinfo->ctrl.driverdata    = cinfo;
cinfo->ctrl.register_appl = b1_register_appl;
cinfo->ctrl.release_appl  = b1_release_appl;
cinfo->ctrl.send_message  = b1_send_message;
cinfo->ctrl.load_firmware = b1_load_firmware;
cinfo->ctrl.reset_ctr     = b1_reset_ctr;
cinfo->ctrl.procinfo      = b1pci_procinfo;
cinfo->ctrl.ctr_read_proc = b1ctl_read_proc;
strcpy(cinfo->ctrl.name, card->name);
SET_MODULE_OWNER(&cinfo->capi_ctrl);

retval = attach_capi_ctr(&capi_ctrl);
```

On the other hand, `kernelcapi.o` provides functions for the drivers to call, most importantly:

```
void
capi_ctr_suspend_output(struct capi_ctr *);

void
capi_ctr_resume_output(struct capi_ctr *);

void
capi_ctr_handle_message(struct capi_ctr *,
                        u16 appl,
                        struct sk_buff *);
```

This interface is similar to the interface an ethernet driver uses to register a `struct net_device`.

Once a driver detects its hardware, it will register the controller using `attach_capi_ctr()`, which corresponds to `register_netdev()`. When an application registers with `kernelcapi.o`, the driver's `register_appl()` method is invoked, analogous to `open()` on a net_device. `kernelcapi.o` will also take care of incrementing the driver module use count before calling into the driver, so that it cannot be unloaded while in use.

Messages to a controller are transferred using the `send_message()` method (equivalent to `hard_start_xmit()`). As in the network stack, calls to `send_message()` shall be serialized, and the driver can call `capi_ctr_{suspend,resume}_output()` to enable / disable receiving messages (corresponding to `netif_{wake,stop}_queue()`). The container used for CAPI messages is the standard Linux socket buffer `struct sk_buff`.

Messages from the controller to an application are sent using `capi_ctr_handle_message()`, corresponding to netif_rx(). This function can be called from hard IRQ context. `kernelcapi.o` will queue the message and hand it on to the addressed application in softint context. For drivers which call `capi_ctr_handle_message()` from non hard IRQ context, it may make sense to add a `_ni` variant which skips this transformation. A driver for a passive card will typically do a lot of processing after receiving a D- or B-channel frame from the hardware, so it will switch to softint processing before even generating the CAPI message to be sent to the application.

The other side of `kernelcapi.o` is the interface to in-kernel CAPI applications, defined in `include/linux/kernelcapi.h` [5]. This interface is constructed to be similar to the userspace CAPI interface, but has been adapted to better fit the general Linux kernel design.

```
u16 capi20_isinstalled(void);
u16 capi20_register(struct capi20_appl *ap);
u16 capi20_release(struct capi20_appl *ap);
u16 capi20_put_message(struct capi20_appl *ap,
          struct sk_buff *skb);
u16 capi20_get_manufacturer(u32 contr,
          u8 buf[CAPI_MANUFACTURER_LEN]);
u16 capi20_get_version(u32 contr,
          struct capi_version *verp);
u16 capi20_get_serial(u32 contr,
          u8 serial[CAPI_SERIAL_LEN]);
u16 capi20_get_profile(u32 contr,
          struct capi_profile *profp);
int capi20_manufacturer(unsigned int cmd,
          void *data);
```

Applications do not use a number (application id) to identify themselves, but rather a `struct capi20_appl`, which is used by `kernelcapi.o` to save data related to that application. When registering, the application provides a callback (`recv_message`) which is called to pass on messages from a controller. The registered `struct capi20_appl` is passed as a parameter on callback, so the application has an easy way of obtaining a handle to its data and state related to the corresponding registration. A callback has been chosen instead of the usual polling or receiving-a-signal method for a pending message, which is not appropriate for kernelspace.

```
ap->private = cdev;
ap->recv_message = capi_recv_message;
cdev->errcode = capi20_register(ap);
```

CAPI4HISAX

There are two essential building parts one finds on all ISDN adapters. First of all, they all provide an electrical interface to an ISDN S0 or U bus (ISDN layer 1). The integrated circuits which implement the physical interface normally also include functionality to provide HDLC framing and collision recognition on the shared D-Channel. On the other hand, of course all ISDN adapters feature an interface to the host computer, be it ISA, PCI, USB or even a parallel port interface.

```
static void * __devinit
probe_st5481(struct usb_device *dev,
             unsigned int ifnum,
             const struct usb_device_id *id)
{
  [...]

  SET_MODULE_OWNER(&adapter->hisax_d_if);
  adapter->hisax_d_if.ifc.priv = adapter;
  adapter->hisax_d_if.ifc.l2l1 = st5481_d_l2l1;

  for (i = 0; i < 2; i++) {
    adapter->bcs[i].adapter = adapter;
    adapter->bcs[i].channel = i;
    adapter->bcs[i].b_if.ifc.priv =
      &adapter->bcs[i];
    adapter->bcs[i].b_if.ifc.l2l1 = st5481_b_l2l1;
    b_if[i] = &adapter->bcs[i].b_if;

  hisax_register(&adapter->hisax_d_if, b_if,
                 "st5481_usb", protocol);
}

[...]

int
hisax_register(struct hisax_d_if *hisax_d_if,
               struct hisax_b_if *b_if[],
               char *name, int protocol)
{
  [...]
  for (i = 0; i < 2; i++)
    b_if[i]->ifc.l1l2 = hisax_b_l1l2;

  hisax_d_if->ifc.l1l2 = hisax_d_l1l2;
  [...]
}
```

FIG. 6: Registering a hardware driver with the HiSax protocol layers

Using these two components, one can build a typical passive ISDN adapter. It provides access to the incoming and outgoing D- and B-Channels at layer 1 level via a FIFO buffer mechanism.

Active ISDN cards provide, in addition to the basic parts, a processor on board, which using its firmware implements the higher layers of the ISDN protocols, like the data link layer (Q.921/LAPD/LAPB) and the application layer (e.g. X.25, Q.931/DSS1 call control and even FAX/modem). In some cases, the firmware furnishes a complete CAPI implementation. For such cards, a driver implementation for the CAPI4Linux subsystem is very straightforward, it is mostly just forwarding messages to and from the card, very similar to what a typical NIC driver does.

The major share of sold ISDN adapters are passive solutions, though. The main reason for this is obviously the price, passive cards cost only a fraction of the active counterparts. When using a passive card, the host processor has to take over the handling of the higher layer B-channel protocols and provide the complete D-channel (call control) handling. This situation seems somewhat analogous to the so called winmodems, which are modems where the host processor has to handle modulation/demodulation instead of an on-board DSP. However, since ISDN is already digital, the effort needed is much smaller.

For a typical application, internet dialup using synchronous PPP (PPP over ISDN), at layer 1 HDLC framing is needed, which is provided by most available ISDN adapters in hardware. Layer 2 and 3 are transparent, the next higher layer is the actual PPP protocol, which is always handled by the host processor. So the only overhead that a passive card has over an active one is caused by usually small FIFO buffers, which increase interrupt load. These FIFOs need typically one IRQ per 32 octets, as opposed to an active card, which only needs one IRQ per packet, which can be up to 2048 octets.

For passive ISDN cards, the complete D-channel / call control handling is provided by the host processor as well. Although not computationally expensive, it needs a proper implementation of the associated state machines, which has been available in the Linux HiSax driver for a long time and even passed certification procedures.

In the Windows world, each manufacturer normally provides their own CAPI drivers for their passive ISDN cards, i.e. the complete implementation from hardware access over call control to high-level protocols like fax. In Linux, all processing above the hardware access is handled by a shared driver, for an open source solution it does not make any sense to duplicate these layers.

```
  switch (adapter->type) {
  case AVM_FRITZ_PCIV2:
    adapter->isac.read_isac
      = &fcpci2_read_isac;
    adapter->isac.write_isac
      = &fcpci2_write_isac;
    adapter->isac.read_isac_fifo
      = &fcpci2_read_isac_fifo;
    adapter->isac.write_isac_fifo
      = &fcpci2_write_isac_fifo;
    break;
  case AVM_FRITZ_PCI:
    adapter->isac.read_isac
      = &fcpci_read_isac;;
    adapter->isac.write_isac
      = &fcpci_write_isac;
    adapter->isac.read_isac_fifo
      = &fcpci_read_isac_fifo;
    adapter->isac.write_isac_fifo
      = &fcpci_write_isac_fifo;
    break;
  }
  isac_setup(&adapter->isac);

[...]

  /* xmit on D-channel */
  hisax_d_if->ifc.l2l1 = isac_d_l2l1;

[...]

static void fcpci_irq(int intno, void *dev,
                      struct pt_regs *regs)
{
  struct fritz_adapter *adapter = dev;
  unsigned char sval;

  sval = inb(adapter->io + 2);
  if (!(sval & AVM_STATUS0_IRQ_ISAC))
          isac_irq(&adapter->isac);
  [...]
}
```

FIG. 7: Usage of the shared ISAC/ISAC-SX driver library code

Splitting HiSax into protocol handling and hardware drivers

The HiSax driver provides the common upper layers as well as hardware access to a wide variety of passive ISDN cards. Whereas traditionally it is built as one monolithic module, during the ongoing restructuring in Linux 2.5, it will be broken down into a shared module for the upper layers and individual modules for hardware access to various kinds of ISDN adapters. Apart from the added flexibility, this ensures better cooperation with current Linux kernel API's, like the PCI/ISAPnP layer or Hotplug capabilities.

For some hardware, the splitting has already been done. As an example, the driver for ST5481 based USB ISDN adapters is available as `hisax_st5481.o`. It registers with the normal `hisax.o` module (see figure 6), which provides ISDN layer 2 and higher, while `hisax_st5481.o` handles layer 1 and hardware access (using the USB API's, naturally).

Another example is the new driver for the AVM Fritz!Card PCI, PCI v2 and PnP cards. Since the hardware on all three of these boards is very similar, they are all driven by one driver, `hisax_fcpcipnp.o`. Again, this driver provides layer 1 and hardware access and leaves the rest to `hisax.o`. The ISDN physical interface and D-channel access on these cards is provided by a Siemens/Infineon chip, the so-called ISAC/ISAC-SX. Since these chips are also used on a variety of other ISDN cards, it would be wasteful to re-implement the needed functionality in every driver. So an additional `hisax_isac.o` module was created which acts as a library providing the methods to drive this kind of chip - as shown in figure 7, the card driver only needs to provide methods to access the hardware, the library will then handle receiving and transmitting on the D-channel.

HiSax interface to CAPI4Linux

Apart from the split-up between hardware / ISDN layer 1 and the upper layers, the protocol implementations of various layer 1 to layer 3 protocols which exist in the HiSax driver today will be continued to be used in the CAPI version of the driver. Adapting HiSax basically means replacing the existing interface to the ISDN4Linux link layer with a CAPI based interface. So `drivers/isdn/hisax/callc.c`, based on superseded assumptions about what features need to be exported, will go away and implementations of the various CAPI objects and state machines will take its place.

We do not intend to describe the new code in great detail here, but rather present a overview of the new files in the current CAPI4HiSax development version and a description of the code contained therein.

- `capi.c` provides the interface and registration with the kernel CAPI layer `kernelcapi.o`.

- `contr.c` provides the implementation of the CAPI *controller* object, which corresponds to one ISDN BRI or PRI interface (of which multiple can exist on one board).

- `appl.c` provides the implementation of the CAPI *application* object, i.e. keeps track of the applications which registered to use CAPI services on a controller.

- `listen.c` provides the implementation of the CAPI *listen* state machine, which is a simple state machine an application uses to determine which kind of notifications it desires to receive (e.g. incoming voice calls).

- `cplci.c plci.c` provides the implementation of the CAPI *PLCI* state machine (see also figure 3). An application always sees a PLCI as exclusively owned, which makes sense - you cannot share one connection between multiple applications. However, at the point where an incoming call is detected but not yet accepted, multiple applications may receive notification of a new PLCI. At the point where one application accepts the calls, it is cleared to the other applications. This is the reason why internally two types of objects exist, one corresponds to the actual incoming call, the other one to the indication of that call to an application. Only after one application has accepted the call and it has been cleared to all others, a one-to-one relationship between this two objects is established.

- `ncci.c` provides the implementation of the CAPI *NCCI* state machine (see also figure 4). Since at this time only non-multiplexed B-channel protocols are supported, there never exists more than one NCCI per PLCI, but the implementation is prepared to lift this restriction in case it becomes necessary.

- `supp_serv.c asn1_address.c asn1_basic_service.c asn1_comp.c asn1_enc.c asn1_aoc.c asn1.c asn1_diversion.c asn1_generic.c` provide a parser for messages exchanged in order to use supplementary services. ASN.1 is a standard which describes how to encode and decode messages into an octet stream and is used in the ISDN standards for the implementation of a variety of supplementary services. Currently, Call Forwarding and Advice of Charge services are supported and can be accessed by applications using the standardized CAPI messages.

The current status of the development version of CAPI4HiSax is basically feature complete and working, but it needs to be adapted and merged with the current code in Linux 2.5, which is still a non-trivial effort.

SUMMARY AND OUTLOOK

We have described the history of the ISDN subsystem in the Linux kernel. ISDN4Linux has done a good job in satisfying the users' most popular needs, in particular synchronous and asynchronous data connections as well as voice service. The internal structure stayed basically unchanged since its introduction in kernel 1.3 and a number of deficiencies have become visible over time.

For the coming stable 2.6 release of the Linux kernel, it is planned to move the ISDN subsystem to a CAPI-centric solution. CAPI is a widely accepted hardware and operating system independent programming interface to basic and supplementary ISDN services. This conversion will overcome the mentioned design problems of ISDN4Linux and the same time introduce a cleaner, smaller and more modern ISDN subsystem in the Linux kernel, moving the policy decisions out to userspace.

The most important milestone in switching to a CAPI based solution is to provide a CAPI implementation for passive ISDN cards. We described how the current HiSax driver will be split into a hardware independent protocol driver which interfaces to submodules driving the actual hardware. We also presented the structure of the current development version, which replaces the interface to the old ISDN4Linux link layer with a new CAPI interface.

By the time that Linux kernel 2.6 is released, the drivers for the active AVM ISDN cards as well as the HiSax driver for passive ISDN cards will have been converted to CAPI4Linux. It is planned to keep the current ISDN4Linux in a working condition to supply continued support for legacy hardware, though it would be good if these drivers would be converted to CAPI as well.

Traditionally, not much userspace code taking advantage of ISDN services except for basic functionality was written for Linux. We hope that with CAPI providing a commonly accept interface, more developers get attracted to implementing Linux based ISDN applications.

There is a lot of room for further improvements and filling in some gaps. In userspace, a distribution independent way of configuring ISDN hardware and network connections would be useful, and a nice graphical adminstration tool would definitely be a nice addition.

`vbox`, an answering machine code, should be adapted to CAPI. The GNU Bayonne project [7] already supports CAPI on Linux, opening the door to IVR (interactive voice response) applications - even elaborate voice mail, call forwarding and callback solutions are possible with this package. Implementing some of the latest CAPI additions, i.e. the switching and conferencing API in the HiSax driver would allow for features otherwise only available on much more expensive specialized telephony boards.

Logging tools (in particular `isdnlog`) should be extended to work directly with a CAPI based system. Also, the implementation of a sophisticated solution as to when to dial-up and disconnect based on various criteria is now possible in userspace.

CAPI4Linux will initially lack some features which were present in ISDN4Linux, like raw IP over ISDN or an AT modem emulator. The former could be implemented with relatively little kernel code, leaving connection setup and teardown to userspace where it is more appropriate. If the needs arises to create a modem emulation, useful for legacy applications that are totally unaware of ISDN and can only deal with modems, it is now possible to implement this nearly completely in userspace.

In conclusion, we are optimistic to see a much improved ISDN subsystem in the Linux kernel 2.6, with the inclusion of an improved kernel CAPI layer and the port of the HiSax driver to provide a CAPI interface to passive ISDN cards and adapters.

[1] Website of the International Telecommunication Union, providing standard documents on ISDN (non-free),
http://www.itu.int

[2] Website of the European Telecommunication Standards Institute. Many useful documents about ISDN are available free of charge,
http://www.etsi.org

[3] Website of the CAPI organization,
http://www.capi.org

[4] CAPI standard documents,
http://www.capi.org/pages/downloads.php

[5] The Linux kernel source, available from
ftp://ftp.kernel.org/pub/linux/kernel/v2.5

[6] Website for ISDN4Linux (unfortunately vastly out of date),
http://www.isdn4linux.de

[7] The Bayonne project,
http://www.gnu.org/software/bayonne/bayonne.html