Kernel CAPI Interface to Hardware Drivers
---

1. Overview

From the CAPI 2.0 specification:
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).

Kernel CAPI operates as a dispatching layer between CAPI applications and CAPI hardware drivers. Hardware drivers register ISDN devices (controllers, in CAPI lingo) with Kernel CAPI to indicate their readiness to provide their service to CAPI applications. CAPI applications also register with Kernel CAPI, requesting association with a CAPI device. Kernel CAPI then dispatches the application registration to an available device, forwarding it to the corresponding hardware driver. Kernel CAPI then forwards CAPI messages in both directions between the application and the hardware driver.

Format and semantics of CAPI messages are specified in the CAPI 2.0 standard. This standard is freely available from http://www.capi.org.

2. Driver and Device Registration

CAPI drivers optionally register themselves with Kernel CAPI by calling the Kernel CAPI function register_capi_driver() with a pointer to a struct capi_driver. This structure must be filled with the name and revision of the driver, and optionally a pointer to a callback function, add_card(). The registration can be revoked by calling the function unregister_capi_driver() with a pointer to the same struct capi_driver.

CAPI drivers must register each of the ISDN devices they control with Kernel CAPI by calling the Kernel CAPI function attach_capi_ctr() with a pointer to a struct capi_ctr before they can be used. This structure must be filled with the names of the driver and controller, and a number of callback function pointers which are subsequently used by Kernel CAPI for communicating with the driver. The registration can be revoked by calling the function detach_capi_ctr() with a pointer to the same struct capi_ctr.

Before the device can be actually used, the driver must fill in the device information fields 'manu', 'version', 'profile' and 'serial' in the capi_ctr structure of the device, and signal its readiness by calling capi_ctr_ready(). From then on, Kernel CAPI may call the registered callback functions for the device.

If the device becomes unusable for any reason (shutdown, disconnect ...), the driver has to call capi_ctr_down(). This will prevent further calls to the callback functions by Kernel CAPI.

3. Application Registration and Communication

Kernel CAPI forwards registration requests from applications (calls to CAPI operation CAPI_REGISTER) to an appropriate hardware driver by calling its register_appl() callback function. A unique Application ID (ApplID, u16) is

allocated by Kernel CAPI and passed to register_appl() along with the parameter structure provided by the application. This is analogous to the open() operation on regular files or character devices.

After a successful return from register_appl(), CAPI messages from the application may be passed to the driver for the device via calls to the send_message() callback function. Conversely, the driver may call Kernel CAPI's capi_ctr_handle_message() function to pass a received CAPI message to Kernel CAPI for forwarding to an application, specifying its ApplID.

Deregistration requests (CAPI operation CAPI_RELEASE) from applications are forwarded as calls to the release_appl() callback function, passing the same ApplID as with register_appl(). After return from release_appl(), no CAPI messages for that application may be passed to or from the device anymore.

4. Data Structures

4.1 struct capi_driver

This structure describes a Kernel CAPI driver itself. It is used in the register_capi_driver() and unregister_capi_driver() functions, and contains the following non-private fields, all to be set by the driver before calling register_capi_driver():

char name[32]
	the name of the driver, as a zero-terminated ASCII string
char revision[32]
	the revision number of the driver, as a zero-terminated ASCII string
int (*add_card)(struct capi_driver *driver, capicardparams *data)
	a callback function pointer (may be NULL)

4.2 struct capi_ctr

This structure describes an ISDN device (controller) handled by a Kernel CAPI driver. After registration via the attach_capi_ctr() function it is passed to all controller specific lower layer interface and callback functions to identify the controller to operate on.

It contains the following non-private fields:

- to be set by the driver before calling attach_capi_ctr():

struct module *owner
	pointer to the driver module owning the device

void *driverdata
	an opaque pointer to driver specific data, not touched by Kernel CAPI

char name[32]
	the name of the controller, as a zero-terminated ASCII string

char *driver_name
	the name of the driver, as a zero-terminated ASCII string

int (*load_firmware)(struct capi_ctr *ctrlr, capiloaddata *ldata)
        (optional) pointer to a callback function for sending firmware and
        configuration data to the device
        Return value: 0 on success, error code on error
        Called in process context.

void (*reset_ctr)(struct capi_ctr *ctrlr)
        (optional) pointer to a callback function for performing a reset on
        the device, releasing all registered applications
        Called in process context.

void (*register_appl)(struct capi_ctr *ctrlr, u16 applid,
                      capi_register_params *rparam)
void (*release_appl)(struct capi_ctr *ctrlr, u16 applid)
        pointers to callback functions for registration and deregistration of
        applications with the device
        Calls to these functions are serialized by Kernel CAPI so that only
        one call to any of them is active at any time.

u16 (*send_message)(struct capi_ctr *ctrlr, struct sk_buff *skb)
        pointer to a callback function for sending a CAPI message to the
        device
        Return value: CAPI error code
        If the method returns 0 (CAPI_NOERROR) the driver has taken ownership
        of the skb and the caller may no longer access it. If it returns a
        non-zero (error) value then ownership of the skb returns to the caller
        who may reuse or free it.
        The return value should only be used to signal problems with respect
        to accepting or queueing the message. Errors occurring during the
        actual processing of the message should be signaled with an
        appropriate reply message.
        May be called in process or interrupt context.
        Calls to this function are not serialized by Kernel CAPI, ie. it must
        be prepared to be re-entered.

char *(*procinfo)(struct capi_ctr *ctrlr)
        pointer to a callback function returning the entry for the device in
        the CAPI controller info table, /proc/capi/controller

const struct file_operations *proc_fops
        pointers to callback functions for the device's proc file
        system entry, /proc/capi/controllers/<n>; pointer to the device's
        capi_ctr structure is available from struct proc_dir_entry::data
        which is available from struct inode.

Note: Callback functions except send_message() are never called in interrupt
context.

- to be filled in before calling capi_ctr_ready():

u8 manu[CAPI_MANUFACTURER_LEN]
        value to return for CAPI_GET_MANUFACTURER

capi_version version
        value to return for CAPI_GET_VERSION

capi_profile profile
        value to return for CAPI_GET_PROFILE

u8 serial[CAPI_SERIAL_LEN]
        value to return for CAPI_GET_SERIAL

4.3 SKBs

CAPI messages are passed between Kernel CAPI and the driver via send_message() and capi_ctr_handle_message(), stored in the data portion of a socket buffer (skb). Each skb contains a single CAPI message coded according to the CAPI 2.0 standard.

For the data transfer messages, DATA_B3_REQ and DATA_B3_IND, the actual payload data immediately follows the CAPI message itself within the same skb. The Data and Data64 parameters are not used for processing. The Data64 parameter may be omitted by setting the length field of the CAPI message to 22 instead of 30.

4.4 The _cmsg Structure

(declared in <linux/isdn/capiutil.h>)

The _cmsg structure stores the contents of a CAPI 2.0 message in an easily accessible form. It contains members for all possible CAPI 2.0 parameters, including subparameters of the Additional Info and B Protocol structured parameters, with the following exceptions:

* second Calling party number (CONNECT_IND)

* Data64 (DATA_B3_REQ and DATA_B3_IND)

* Sending complete (subparameter of Additional Info, CONNECT_REQ and INFO_REQ)

* Global Configuration (subparameter of B Protocol, CONNECT_REQ, CONNECT_RESP and SELECT_B_PROTOCOL_REQ)

Only those parameters appearing in the message type currently being processed are actually used. Unused members should be set to zero.

Members are named after the CAPI 2.0 standard names of the parameters they represent. See <linux/isdn/capiutil.h> for the exact spelling. Member data types are:

u8   for CAPI parameters of type 'byte'

u16   for CAPI parameters of type 'word'

u32   for CAPI parameters of type 'dword'

_cstruct   for CAPI parameters of type 'struct'
            The member is a pointer to a buffer containing the parameter in CAPI encoding (length + content). It may also be NULL, which will be taken to represent an empty (zero length) parameter.

Subparameters are stored in encoded form within the content part.

_cmstruct alternative representation for CAPI parameters of type 'struct'
(used only for the 'Additional Info' and 'B Protocol' parameters)
The representation is a single byte containing one of the values:
CAPI_DEFAULT: The parameter is empty/absent.
CAPI_COMPOSE: The parameter is present.
Subparameter values are stored individually in the corresponding
_cmsg structure members.

Functions capi_cmsg2message() and capi_message2cmsg() are provided to convert messages between their transport encoding described in the CAPI 2.0 standard and their _cmsg structure representation. Note that capi_cmsg2message() does not know or check the size of its destination buffer. The caller must make sure it is big enough to accomodate the resulting CAPI message.

5. Lower Layer Interface Functions

(declared in <linux/isdn/capilli.h>)

```
void register_capi_driver(struct capi_driver *drvr)
void unregister_capi_driver(struct capi_driver *drvr)
```
register/unregister a driver with Kernel CAPI

```
int attach_capi_ctr(struct capi_ctr *ctrlr)
int detach_capi_ctr(struct capi_ctr *ctrlr)
```
register/unregister a device (controller) with Kernel CAPI

```
void capi_ctr_ready(struct capi_ctr *ctrlr)
void capi_ctr_down(struct capi_ctr *ctrlr)
```
signal controller ready/not ready

```
void capi_ctr_suspend_output(struct capi_ctr *ctrlr)
void capi_ctr_resume_output(struct capi_ctr *ctrlr)
```
signal suspend/resume

```
void capi_ctr_handle_message(struct capi_ctr * ctrlr, u16 applid,
                             struct sk_buff *skb)
```
pass a received CAPI message to Kernel CAPI
for forwarding to the specified application

6. Helper Functions and Macros

Library functions (from <linux/isdn/capilli.h>):

```
void capilib_new_ncci(struct list_head *head, u16 applid,
                      u32 ncci, u32 winsize)
void capilib_free_ncci(struct list_head *head, u16 applid, u32 ncci)
void capilib_release_appl(struct list_head *head, u16 applid)
void capilib_release(struct list_head *head)
void capilib_data_b3_conf(struct list_head *head, u16 applid,
                      u32 ncci, u16 msgid)
u16  capilib_data_b3_req(struct list_head *head, u16 applid,
                      u32 ncci, u16 msgid)
```

Macros to extract/set element values from/in a CAPI message header
(from <linux/isdn/capiutil.h>):

Get Macro	Set Macro	Element (Type)
CAPIMSG_LEN(m)	CAPIMSG_SETLEN(m, len)	Total Length (u16)
CAPIMSG_APPID(m)	CAPIMSG_SETAPPID(m, applid)	ApplID (u16)
CAPIMSG_COMMAND(m)	CAPIMSG_SETCOMMAND(m,cmd)	Command (u8)
CAPIMSG_SUBCOMMAND(m)	CAPIMSG_SETSUBCOMMAND(m, cmd)	Subcommand (u8)
CAPIMSG_CMD(m)	-	Command*256 + Subcommand (u16)
CAPIMSG_MSGID(m)	CAPIMSG_SETMSGID(m, msgid)	Message Number (u16)
CAPIMSG_CONTROL(m)	CAPIMSG_SETCONTROL(m, contr)	Controller/PLCI/NCCI (u32)
CAPIMSG_DATALEN(m)	CAPIMSG_SETDATALEN(m, len)	Data Length (u16)

Library functions for working with _cmsg structures
(from <linux/isdn/capiutil.h>):

unsigned capi_cmsg2message(_cmsg *cmsg, u8 *msg)
        Assembles a CAPI 2.0 message from the parameters in *cmsg, storing the result in *msg.

unsigned capi_message2cmsg(_cmsg *cmsg, u8 *msg)
        Disassembles the CAPI 2.0 message in *msg, storing the parameters in *cmsg.

unsigned capi_cmsg_header(_cmsg *cmsg, u16 ApplId, u8 Command, u8 Subcommand,
                          u16 Messagenumber, u32 Controller)
        Fills the header part and address field of the _cmsg structure *cmsg with the given values, zeroing the remainder of the structure so only parameters with non-default values need to be changed before sending the message.

void capi_cmsg_answer(_cmsg *cmsg)
        Sets the low bit of the Subcommand field in *cmsg, thereby converting _REQ to _CONF and _IND to _RESP.

char *capi_cmd2str(u8 Command, u8 Subcommand)
        Returns the CAPI 2.0 message name corresponding to the given command and subcommand values, as a static ASCII string. The return value may be NULL if the command/subcommand is not one of those defined in the CAPI 2.0 standard.

7. Debugging

The module kernelcapi has a module parameter showcapimsgs controlling some debugging output produced by the module. It can only be set when the module is loaded, via a parameter "showcapimsgs=<n>" to the modprobe command, either on the command line or in the configuration file.

If the lowest bit of showcapimsgs is set, kernelcapi logs controller and application up and down events.

In addition, every registered CAPI controller has an associated traceflag parameter controlling how CAPI messages sent from and to tha controller are logged. The traceflag parameter is initialized with the value of the showcapimsgs parameter when the controller is registered, but can later be changed via the MANUFACTURER_REQ command KCAPI_CMD_TRACE.

If the value of traceflag is non-zero, CAPI messages are logged.
DATA_B3 messages are only logged if the value of traceflag is > 2.

If the lowest bit of traceflag is set, only the command/subcommand and message length are logged. Otherwise, kernelcapi logs a readable representation of the entire message.