DVB-Multicast-Client API-Specification

Date: 17.07.2009

Version: 2.00

Author: Deti Fliegl <fliegl@baycom.de>

This Document describes the API provided by the DVB-Multicast-Client library

Receiver API

Module global functions

Function: `int recv_init(char *interface, int port);`

Description: Initializes the Multicast IP-Receiver API.

Arguments: **interface:** name of network interface to operate on, can be empty string

port: port number for IP service, use 23000 for NetCeiver operation

Returns: 0 on success

Function: `int recv_exit(void);`

Description: Cleanup IP-Receiver API

Arguments: -

Returns: 0 on success

Receiver Handling

Function: `recv_info_t *recv_add (void);`

Description: Add a new receiver instance

Arguments: -

Returns: A pointer to a receiver instance on success

NULL on error

Function: `void recv_del (recv_info_t *receiver);`

Description: Delete a receiver instance

Arguments: **receiver:** Pointer to receiver instance

Returns: -

Function: `int register_ts_handler (recv_info_t * receiver, void *function, void *context);`

Description: Register a function for handling TS data

Arguments: **receiver:** A previously allocated receiver instance

function: A pointer to a function of type `int ts_handler(unsigned char *buffer, size_t len, void *context)`,

can be NULL to unregister handler

context: A pointer to same data that is being used as context in handler

Returns: 0 on success

Function: `int register_ten_handler (recv_info_t * receiver, void *function, void *context);`

Description: Register a function for handling TEN data

Arguments: **receiver:** A previously allocated receiver instance

function: A pointer to a function of type `int ten_handlertra_t *ten, void *context)`,

can be NULL to unregister handler

context: A pointer to same data that is being used as context in handler

Returns: 0 on success

Handler Functions

Function: `int handle_ts (unsigned char *buffer, size_t len, void *context)`

Description: Handle incoming TS packets

Arguments: **buffer:** a buffer containing TS packets

len: length of the buffer

context: reference to context data previously registered by `register_ts_handler`

Returns: number of successfully processed bytes (should be len).

Function: `int handle_ten (tra_t *ten, void *context)`

Description: Handle incoming TEN information

Arguments: **ten:** structure containing all signal information

context: reference to context data previously registered by `register_ten_handler`

Returns: 0 on success

Tuning

Function: `int recv_tune (recv_info_t *receiver, fe_type_t type, int satpos, dvblo_sec_t *sec, struct dvb_frontend_parameters *fe_parms, dvb_pid_t *pids);`

Description: Tune receiver instance to given parameters

Arguments: **receiver:** previously allocated receiver instance

type: frontend type (DVB-S: FE_QPSK, DVB-C: FE_QAM, DVB-T: FE_OFDM)

satpos: satellite position for DVB-S in 10th degrees + 1800

Example: 19,2°E = 192 +1800 = 1992

sec: Satellite Equipment Control data for DVB-S operation

When satellite position is given only `sec->voltage` has to be set to `SEC_VOLTAGE_13` or `SEC_VOLTAGE_18`

When no satellite position is given `mini_cmd and tone_mode` have to be set.

fe_parms: tuning parameters according to linux kernel structure `dvb_frontend_parameters`.

For DVB-S a direct transponder frequency in kHz has to be given. If no satellite position is given a IF frequency can be used.

For DVB-C/T a direct transmitter frequency in Hz has to be given.

pids: a list of PIDs that should be streamed to the receiver.

The list has to be terminated by a pid value of -1.

Returns: 0 on success

PID-Handling

Function: `int recv_pid_add (recv_info_t * receiver, dvb_pid_t *pid);`

Description: Add a single PID to a receiver instance

Arguments: **receiver:** previously allocated receiver instance

pid: a single PID to be added.

If no conditional access (CA) is being used set `id` element of `dvb_pid_t` structure to 0. In CA operation `id` must be set to the PROGRAM_NUMBER (aka SID) of the corresponding program the PIDs belong to.

Returns: 0 on success

Function: `int recv_pid_del (recv_info_t * receiver, int pid);`

Description: Remove a single PID from a receiver instance

Arguments: **receiver:** previously allocated receiver instance

pid: a single PID to be removed.

Returns: 0 on success

Function: `int recv_pids (recv_info_t * receiver, dvb_pid_t *pids);`

Description: Set a list of PIDs, replacing all existing PIDs

Arguments: **receiver:** previously allocated receiver instance

pid: a list of PIDs to be added. The list has to be terminated by a pid value of -1.

Returns: 0 on success

Function: `int recv_stop (recv_info_t * receiver);`

Description: Remove all PIDs from a receiver instance.

Arguments: **receiver:** previously allocated receiver instance.

Returns: 0 on success.

NetCeiver Discovery

The NetCeiver Discovery is being used to find out which NetCeivers are available with their individual configuration. The discovery is a background process started by the `recv_init` function.

Function: `void nc_lock_list (void);`

Description: Locks the internal list of active NetCeivers to avoid modification by the discovery process.

Arguments: -

Returns: -

Function: `void nc_unlock_list (void);`

Description: Unlocks the internal list of active NetCeivers to allow modification by the discovery process.

Arguments: -

Returns: -

Function: `netceiver_info_list_t *nc_get_list (void);`

Description: Returns a pointer to a list of discovered NetCeivers.

Arguments: -

Returns: A list of discovered NetCeivers.

Example code to traverse the available NetCeivers and tuner slots

```
int n, i;
recv_init (NULL, 0);
netceiver_info_list_t *nc_list = nc_get_list ();
nc_lock_list ();
for (n = 0; n < nc_list->nci_num; n++) {
     netceiver_info_t *nci = nc_list->nci + n;
     printf ("Found NetCeiver: %s \n", nci->uuid);
     for (i = 0; i < nci->tuner_num; i++) {
          printf ("  Tuner: %s, Type %d\n",
          nci->tuner[i].fe_info.name,
          nci->tuner[i].fe_info.type);
     }
}
nc_unlock_list ();
recv_exit();
```

MLD Reporter

Function: `void mld_client_init (char *intf);`
Description: Start optional MLDv2 client to make zapping faster and more reliable
Arguments: **intf:** name of interface to operate on.
Returns: -

Function: `void mld_client_exit (void);`
Description: Stop MLDv2 client.
Arguments: -
Returns: -

MMI Client

The MMI Client allows access to the MMI functions of CAMs plugged into a NetCeiver. The MMI allows interactive dialogs where users can select menu items, enter data or simply get notified by a text message.

Function: `UDPContext *mmi_broadcast_client_init(int port, char *iface);`

Description: Start a client process for receiving MMI broadcasts that are caused by a MMI session initiated by the CAM. Such a session could be used by the CAM to inform the user of a program that cannot be decrypted or to ask for a PIN code.

Arguments: **interface:** name of network interface to operate on, can be empty string

port: port number for IP service, use 23000 for NetCeiver operation, can be 0

Returns: A context structure or NULL if the call failed.

Function: `int mmi_poll_for_menu_text(UDPContext *s, mmi_info_t *m, int timeout);`

Description: returns text of MMI session initiated by the CAM and received via broadcast.

Arguments: **s** a context returned by mmi_broadcast_client_init.

m a pointer to a mmi_info_t data structure which will be filled by the function on reception of a broadcast message.

timeout a value in ms to wait blocking for a message.

Returns: a value > 0 if a message was received.

Function: `void mmi_broadcast_client_exit(UDPContext *s);`

Arguments: **s** a UDPContext previously returned by `mmi_broadcast_client_init`

Description: End a previously started client process for receiving MMI broadcasts.

Returns: -

Function: `int mmi_open_menu_session(char *uuid, char *iface, int port, int slot);`

Description: Opens a user initiated MMI session to a CAM in a NetCeiver given by the UUID argument.

Arguments: **uuid:** specifies a NetCeiver by its UUID

iface: name of network interface to operate on, can be empty string

port: port number for IP service, use 23013 for NetCeiver operation, can be 0

slot: use 0 for 1st CAM slot, use 1 for 2nd CAM slot

Returns: A handle to the newly opened MMI session or -1 if the call failed. If t

Function: `int mmi_cam_reset(char *uuid, char *intf, int port, int slot);`

Description: Issue a reset on a specific slot of a given NetCeiver.

Arguments: **uuid**: specifies a NetCeiver by its UUID

intf: name of network interface to operate on, can be empty string

port: port number for IP service, use 23013 for NetCeiver operation, can be 0

slot: use 0 for 1st CAM slot, use 1 for 2nd CAM slot

Returns: 0 on success

Function: `int mmi_send_menu_answer(int s, char *buf, int buf_len);`

Description: initiates a MMI session when buf_len is 0 or sends text to an already opened one.

Arguments: **s:** specifies a previously allocated session context by

`mmi_open_menu_session`

`buf:` specifies an ISO 8859-1 coded text buffer

buf_len: sets the length of the text that should be sent to the CAM

Returns: 0 on success

Function: `int mmi_get_menu_text(int sockfd, char *buf, int buf_len, int timeout);`

Description: Gets text from a opened MMI session.

Arguments: **s:** specifies a previously allocated session context by

buf: specifies a buffer to receive a ISO 8859-1 coded text from the CAM

buf_len: sets the size of the buffer

Returns: a value >0 when a message was received.

Function: `int mmi_close_menu_session(int handle);`

Description: Closes a previously opened MMI session

Arguments: **handle:** returned by `mmi_open_menu_session.`

Returns: -