RadiSys.

Using the Digital Broadcast Environment Pak

Version 2.1

www.radisys.com
World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044
International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620
RadiSys Microware Communications Software Division, Inc.
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Revision A
November 2001

Copyright and publication information

This manual reflects version 2.1 of the Digital Broadcast Environment Pak.
Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

Table of Contents

Chapter 3: The Navigation API 77

Chapter 4: The MPEG Private Data API 213

Chapter 5: The DBE Pak Device Driver Architecture 241

Chapter 6: The Channel Manager Protocol Driver 255

Chapter 7: The Tuner Device Driver 259

Chapter 1: Getting Started

This chapter is an overview of the Digital Broadcast Environment Pak. It explains the architecture, product functions, and modules provided in this package.

Introduction

The Digital Broadcast Environment (DBE) Pak provides extensions to Microware's DAVID and DAVIDLite operating environments for supporting digital broadcast television applications in consumer devices.

From an application's perspective, the package provides Application Programming Interfaces (APIs) for channel navigation, for retrieval of Program Specific Information (PSI), System or Service Information (SI), and private data from an incoming MPEG-2 transport stream.

The package also includes system-level software for reading and parsing an ATSC (Advanced Television Systems Committee) or DVB (Digital Video Broadcast) compliant bit-stream in order to extract and build the list of available television channels.

In addition the package includes support for EPG applications to extract the TV schedule information from the SI tables. Parental Control functionality is also included in the package.

The DBE Pak also includes template SoftStax device drivers for managing tuner and conditional access hardware. In an actual digital broadcast consumer product, these device drivers must be modified and ported to the vendor's specific hardware. However, the source code for the template drivers included with the package illustrates the interfaces actual drivers must support. If in a particular environment the conditional access component is not needed, it can be left out of the system. This will not affect the rest of the modules.

Finally, the DBE Pak includes updated versions of Microware's Player Shell, Real-Time Network Driver, and DUXMAN MPEG Demultiplexer Driver to support the new features required for a digital broadcast television environment.

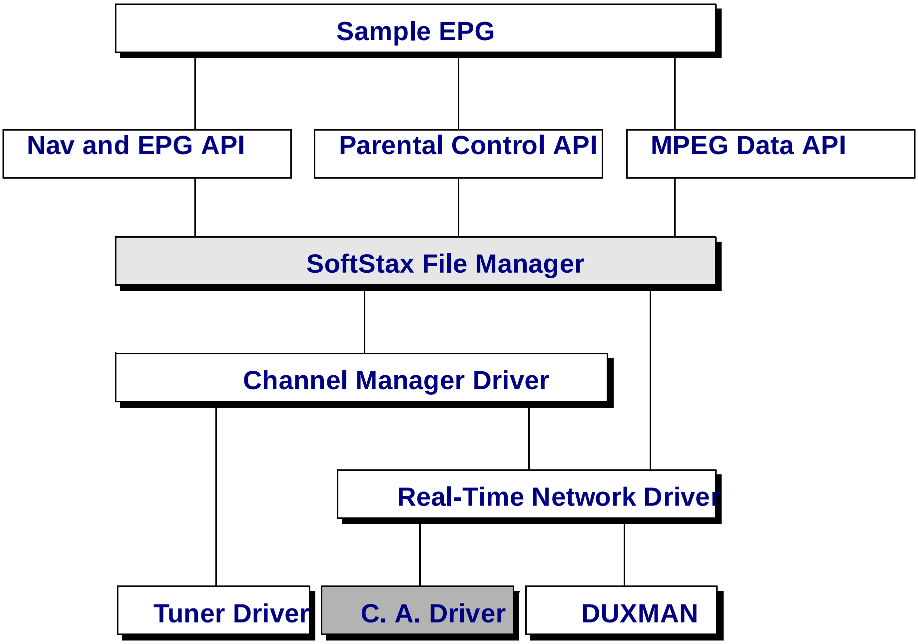
DBE Pak Requirements

The DBE Pak works in conjunction with the SoftStax Base Pak software to provide an environment for creating digital broadcast television consumer devices.

DBE Pak Architecture

Figure 1-1 Digital Broadcast Environment Pak Architecture shows the architecture and organization of the software modules provided in the DBE Pak. The SoftStax File Manager is shaded since it isn't included as part of the package. The Conditional Access driver is shaded because it is an optional component in the system.

Figure 1-1 Digital Broadcast Environment Pak Architecture



The function of each of these software components is described briefly below.

Sample EPG

A sample EPG is provided with the DBE Pak as an example of how to use the Navigation and EPG API. It provides a Grid based User Interface to display the TV programming schedule. It also monitors the input device and performs requested channel change operations. Information about the current event is displayed when the channel is changed.

Navigation API

The Navigation API provides an easy-to-use programming interface to the Channel Manager Protocol Driver. Through the use of the Navigation API, application programs can perform channel change operations, obtain information on available channels, and organize channels into logical channel rings.

Channel rings provide a simple, yet powerful, method for organizing the potentially hundreds of channels available in a digital broadcast television environment into user-defined channel lists. For example, each member of a family could define a channel ring consisting of his or her favorite channels. When the channel-up or channel-down button is pressed, an application can then traverse the channels in the current channel ring through the use of the Navigation API.

For More Information

For more information on the Navigation API, see *Chapter 2: Using the DBE APIs* and *Chapter 3: The Navigation API*.

EPG API

The EPG API is part of the Navigation API. It supports an EPG application by allowing it to retrieve information from the Channel Manager driver about events playing on specific channels at a specific time. The API presents the application with an easy to use interface for obtaining information about available events on the network, organizing the events into a configurable cache, and managing the memory associated with the cache.

For More Information

For more information on the Navigation API, see *Chapter 2: Using the DBE APIs* and *Chapter 3: The Navigation API*.

Parental Control API

The Parental Control API allows the end-user to restrict Television viewing by members of the household. The API provides multiple users with access to the SetTop Box via password protected user accounts. Access may be restricted based on time of the day, rating of the event and the channel number.

For More Information

For more information on the Parental Control API, see *Chapter 2: Using the DBE APIs* and Chapter 4: The Parental Control API.

MPEG Private Data API

The MPEG Private Data API provides a mechanism to allow application programs to request and retrieve PSI, SI, and private data from the incoming MPEG-2 transport-stream. This feature can be used by Electronic Program Guide (EPG) applications, as well as other types of applications. Filtering criteria can be specified so only information of interest is passed to the requesting application.

The MPEG Private Data API calls to request information from the MPEG-2 stream are asynchronous. These calls return control to the application before the requested data has been retrieved from the network. This allows application programs to register multiple outstanding requests for various tables from an incoming bit-stream with the MPEG Private Data API. The API also provides mechanisms to allow applications to register for an asynchronous notification to be sent to the application when requested data is available to be read.

For More Information

For more information on the MPEG Private Data API, see *Chapter 2: Using the DBE APIs* and *Chapter 4: The MPEG Private Data API.*

Channel Manager Protocol Driver

The Channel Manager Protocol Driver monitors and parses the appropriate tables in the MPEG-2 transport stream to collect the list of available channels and the tuning parameters (examples: frequency and MPEG-2 program number) associated with each channel. The channel list built by the driver is made available to application programs and API libraries to support channel change operations.

Depending on which version of the package is being used, the Channel Manager Protocol Driver supports either the ATSC-SI standard or the DVB-SI standard.

For More Information

For more information on the Channel Manager Protocol Driver, see *Chapter 5: The DBE Pak Device Driver Architecture* and *Chapter 6: The Channel Manager Protocol Driver*.

Tuner Device Driver

The Tuner Device Driver is responsible for programming the tuner hardware to tune to a specific frequency when instructed to do so by the Channel Manager Protocol Driver.

Source code for a template Tuner Device Driver is included with the DBE Pak. This template Tuner Device Driver does not support any specific tuner hardware, but instead serves as a skeleton driver which can be completed according to the requirements of the system's actual tuner hardware.

For More Information

For more information on the Tuner Device Driver, see *Chapter 5: The DBE Pak Device Driver Architecture* and *Chapter 7: The Tuner Device Driver*.

Conditional Access Device Driver

The Conditional Access Device Driver is responsible for programming the conditional access hardware to decrypt a specific MPEG-2 program or service when instructed to do so by the Real-Time Driver. This is an optional component and may be omitted without affecting the rest of the system.

Source code for a template Conditional Access Device Driver is included with the DBE Pak. This template Conditional Access Device Driver does not support any specific conditional access hardware, but instead serves as a skeleton driver which can be completed according to the requirements of the system's actual conditional access hardware.

For More Information

For more information on the Conditional Access Device Driver, see *Chapter 5: The DBE Pak Device Driver Architecture* and *Chapter 8: The Conditional Access Device Driver*.

Real-Time Network Driver

The Real-Time Network Driver provides several different functions to application programs and other drivers in the DBE Pak architecture. By using the Real-Time Network Driver, other components in the system can:

- register requests to obtain PSI, SI, or private data from the incoming MPEG-2 transport stream;
- start or stop the playout of an MPEG-2 program;
- set user preferences for language selection in programs with multiple audio tracks; and
- request information on a specific program in the currently tuned transport stream.

The Real-Time Network Driver you receive with DBE 1.1 has been enhanced since the release of DAVID 2.1.

For More Information

For more information on the Real-Time Device Driver, see *Chapter 5: The DBE Pak Device Driver Architecture* and *Chapter 9: The Real-Time Network Driver*. Also see ***SPF 2.1 Porting Guide*** in the DAVID 2.1 manual set.

DUXMAN

DUXMAN manages the MPEG-2 packet demultiplexer hardware. In the DAVID architecture, DUXMAN is used by both the Softstax Real-Time Network Driver to read PSI, SI, and private data sections; and by the MPFM file manager to control the play-out of MPEG-2 programs.

In the DBE Pak's software, the Real-Time Network Driver issues requests to DUXMAN to obtain tables and private data conforming to the MPEG-2 section syntax from the incoming MPEG-2 transport stream. DUXMAN collects the packets for the requested information, assembles the collected packets into MPEG-2 sections, and when a complete section arrives passes it to the Real-Time Network Driver.

Application programs typically do not interface directly to DUXMAN, but instead either use the appropriate API calls or make direct calls to the Channel Manager Protocol Driver or Real-Time Network Driver. The DUXMAN you receive with DBE 1.1 has been enhanced since the release of DAVID 2.1

For More Information

For a discussion of DUXMAN, refer to the ***Using DUXMAN 2.1*** manual in the DAVID 2.1 manual set.

Chapter 2: Using the DBE APIs

This chapter presents an overview of writing applications using the Navigation, EPG, Parental Control and MPEG Private Data API calls.

RadiSys.

MICROWARE SOFTWARE

The Navigation API

Purpose of the API

The Navigation API presents an easy-to-use interface for obtaining information about available channels, changing channels, and organizing channels in a user-friendly manner to application programs.

Architecture of the API

The Navigation API is implemented as a set of function calls whose object code is contained in a statically-linked library named `nav_api.l`. Applications issuing calls to the Navigation API must be linked to this library.

Since the Navigation API calls are provided by a library linked to application programs, the calls execute in user-state in the context of the calling process. However, several of the API calls return pointers to data structures owned and maintained by the Channel Manager Protocol Driver. Some of these data structures contain information describing the set of available channels, and are typically created and updated by the Channel Manager Protocol Driver as it parses the System or Service Information tables received from the network. Other data structures contain information about favorite channel lists, and are typically created and updated by the Channel Manager Protocol Driver in response to calls issued by the Navitation API itself. The complete set of data structures created and maintained by the Channel Manager Protocol Driver is collectively referred to as the *channel map*.

The Navigation API provides function calls for implementing most channel managerment functions; thus, a typical application may never need to directly access fields in any of the channel map's data structures. For example, Navigation API calls are available for creating and maintaining favorite channel lists, selecting the current favorite channel list, changing channels to a specified channel number, and changing channels to the next or preceeding channel in the current channel list.

However, in some cases an application may wish to directly examine certain fields in a channel map data structure. To allow for this, the Channel Manager Protocol Driver sets the memory protection permissions on these data structures so applications can read the structures, but are prevented from writing to them.

Before describing an overview of using the Navigation API calls, an overview of these channel map data structures is provided.

Channel Map Data Structures

Three primary types of data structures are maintained by the Channel Manager Protocol Driver but made available to application programs: the `ring` structure, the `channel` structure and the `chan_info` structure. These three types of structures are defined in the file `<SPF/nav_api.h>` and further defined below.

The Ring Structure

The `ring` structure provides a mechanism for applications to organize the available television channels into lists known as *channel rings*. A channel ring represents a doubly linked list of `channel` structures. Each `channel` structure in a ring corresponds to one television channel from the set of all available channels. The Channel Manager maintains a separate ring structure for each channel ring. Each ring structure contains a head and tail pointer to the list of channel structures for the ring, as well as other fields used for housekeeping purposes.

Note

Although the term *channel ring* implies the channels in a ring may be stored in a circular list, this in fact is not the case. However, during channel-up and channel-down operations the Navigation API and Channel Manager Protocol Driver automatically move from one end of the current channel ring to the other in order to present the appearance of a circular list to applications.

Main Channel Ring

The Channel Manager Protocol Driver automatically creates a *Main Channel Ring* from the data in the ATSC System Information or DVB Service Information (SI) Tables of the incoming MPEG-2 transport streams. This channel ring contains the total collection of channels defined by the SI tables for all of the frequencies which the TV or set-top box can receive.

Note

Although DVB compliant Service Information Tables may also describe channels available in other networks, any such channel definitions are ignored by the DVB Channel Manager Protocol Driver.

Application programs can use the Navigation API to examine the channels in the Main Channel Ring and create additional favorite channel rings containing a subset of these channels. Thus, a television channel may exist in multiple channel rings.

Note

For some digital broadcast enviornments the set of channels transmitted on each frequency can change dynamically. For example, an ATSC terrestrial broadcaster may broadcast a single high-definition television channel during prime-time hours and several standard-definition television channels during other parts of the day. For such environments the Channel Manager will not become aware of changes to the channel list for a specific frequency until it is tuned to that frequency. Thus the set of channels listed in the main channel ring may not always accurately reflect the current list of available channels for each frequency. However during channel change operations the Channel Manager will always first tune to the proper frequency and then validate or update its channel information for that frequency before selecting the channel to be decoded.

Favorite Rings

Although channel rings can be created for any purpose, one of their most common uses is to define the list of favorite channels for a viewer. An application using the DBE Pak can present the list of all available channels to the viewer and allow the viewer to select those channels he or she would like to add to a favorite channel ring. Multiple favorite channel rings could be created, one for each member of a family. Calls provided by the Navigation API could then be used to select a favorite channel ring and traverse the channels within this ring.

The individual `ring` structures contained in the channel map are organized into a doubly-linked list. The first `ring` structure in the list is always the main channel ring; the remaining `ring` structures represesent favorite channel rings.

The Channel and Chan_info Structures

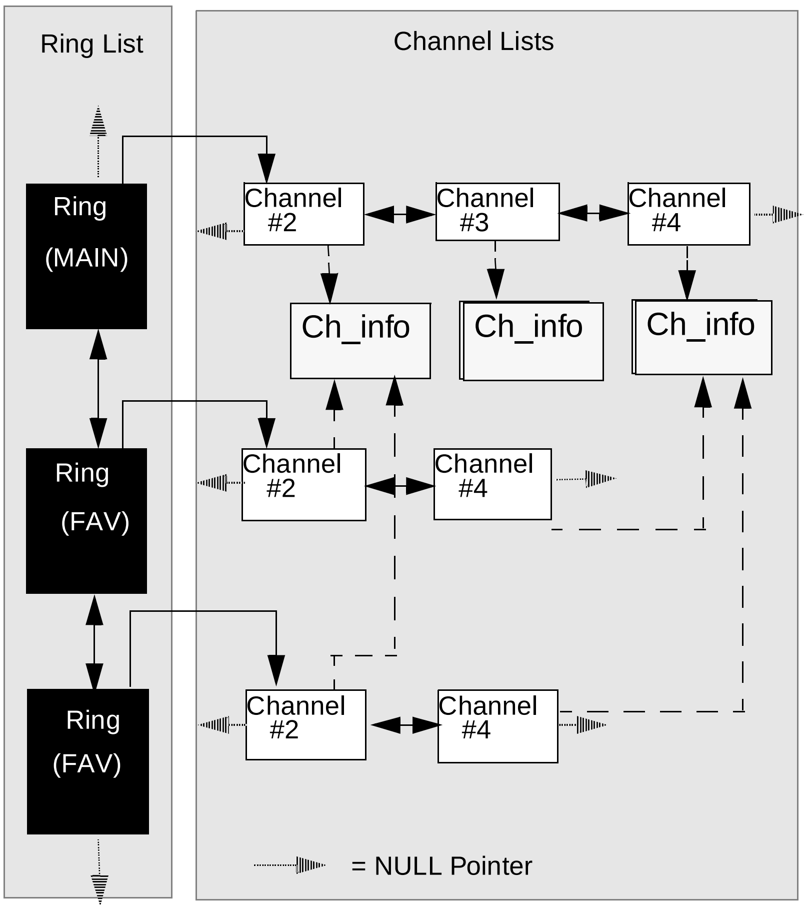
The `channel` and `chan_info` structures contain information on an individual television channel. The `channel` structure contains next and previous pointers for traversing a channel ring and a pointer to the channel's `chan_info` structure. The `chan_info` structure is created by the Channel Manager Protocol Driver from information contained in the appropriate MPEG-2 SI tables, and contains deatailed information about the channel.

A single `chan_info` structure may be referenced by several `channel` structures in different channel rings. For example, a `channel` structure in a favorite channel ring may point to the same `chan_info` structure as a `channel` structure in the Main Channel Ring or a different favorite channel ring.

Relationship between ring, channel and chan_info structures

The relationship between `ring` structures, `channel` structures, and `chan_info` structures is illustrated in Figure 2-1, Channel Map Data Structures.

Figure 2-1 Channel Map Data Structures



Declaration

This `ring` structure is declared in the file `SPF/nav_api.h` as follows:

```
typedef struct ring    ring;
typedef struct ring    *Ring;
typedef struct channel channel;
typedef struct channel *Channel;

#defineRING_MAX_NAME32

typedef struct ring
{
   Ring        ring_next;
   Ring        ring_prec;
   char        ring_name[RING_MAX_NAME];
   u_int32     ring_id;
   Channel     ring_chan_list_head;
   Channel     ring_chan_list_tail;
   Channel     ring_curr_chan;
   u_int32     ring_curr_major_chan_num;
   u_int32     ring_curr_minor_chan_num;
   u_int32     ring_prev_major_chan_num;
   u_int32     ring_prev_minor_chan_num;
   u_int16     ring_num_channels;
   u_char      ring_type;
   u_char      ring_rsvd;
};
```

Fields

The fields in the `ring` structure are defined as follows:

`ring_next` and `ring_prev`

The currently defined `ring` structures are stored as a doubly-linked list of channel rings. This list may be traversed using the `ring_next` and `ring_prec` pointers (see the figure *Channel Ring Data Structure* on page 29.) The two ends of the list are NULL-terminated.

`ring_name`
: is a unique name assigned to a ring. In the DVB environment, for the Main Channel Ring created by the Channel Manager Protocol Driver the `ring_name` field is obtained from the Network Name Descriptor in the Network Information Table. In the ATSC environment, the name of the Main Channel Ring is defined in the device descriptor for the Channel Manager Driver. For a channel ring created by an application, the name is defined by the application creating the ring.

`ring_id`
: is a value assigned by the Channel Manager Protocol Driver to uniquely identify a given channel ring. The Main Channel Ring always has a `ring_id` of zero.

`ring_chan_list_head`
: points to the head of the rings's channel list.

`ring_chan_list_tail`
: points to the tail of the rings's channel list.

`ring_curr_chan`
: points to the ring's current channel.

`ring_curr_major_chan_num`
: is the major channel number for the ring's current channel.

`ring_curr_minor_chan_num`
: is the minor channel number for the ring's current channel.

`ring_prev_major_chan_num`

is the major channel number for the ring's previously active channel.

`ring_prev_minor_chan_num`

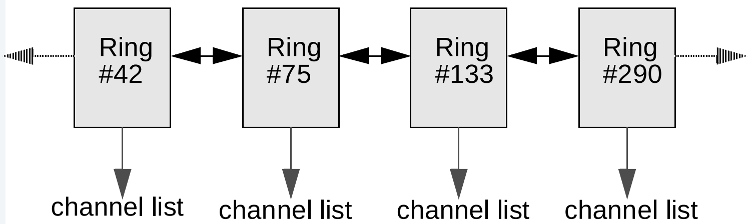
is the minor channel number for the ring's previously active channel.

`ring_num_channels` specifies the number of channels in the ring.

`ring_type` defines the type of the channel ring. For the Main Channel Ring, the `ring_type` is set to `RING_TYPE_MAIN.` For a channel ring created by an application, the `ring_type` is set to `RING_TYPE_FAVORITE`.

`ring_rsvd` is reserved for future use.

Figure 2-2 Channel Ring Data Structure



channel

Declaration

The `channel` structure is declared in the file `SPF/nav_api.h` as follows:

```
typedef struct channelchannel;
typedef struct channel*Channel;

typedef struct channel
{
  Channel   ch_next;
  Channel   ch_prec;
  Chan_info ch_info;
};
```

Fields

The fields in the `channel` structure are defined as follows:

`ch_next and ch_prec`
: A channel ring is a doubly-linked list of `channel` structures. The list may be traversed using the `ch_next` and `ch_prec` pointers. The two ends of the list are NULL-terminated.

`ch_info`
: is a pointer to a structure that contains detailed information about the channel. This structure may be shared by channels across multiple rings. For example, if two channels in two different rings correspond to the same viewer perceived channel, then they will point to the same `ch_info` structure.

Declaration

The `chan_info` structure is declared in the file `SPF/nav_api.h` as follows:

```
typedef struct chan_infochan_info;
typedef struct chan_info*Chan_info;

typedef struct chan_info
{
   u_int32    major_chan_num;
   u_int32    minor_chan_num;
   void       *chan_user;
   u_char     chan_flags;
   u_char     chan_nbr_fav_rings;
   u_char     chan_rsvd[2];
};
```

Fields

The fields in the `ch_info` structure are defined as follows:

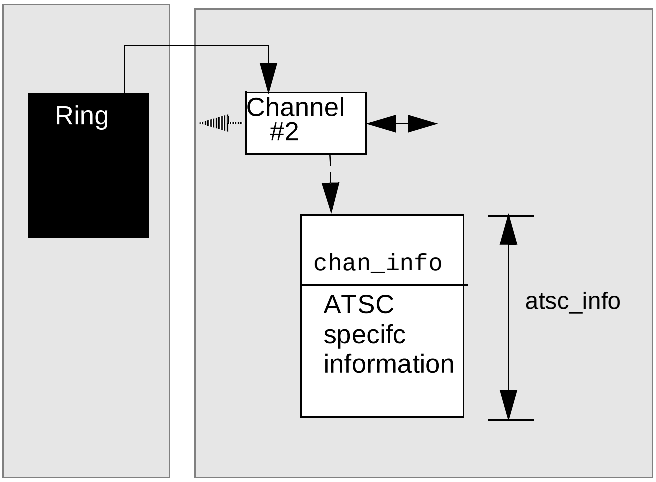
`major_chan_num`	In the ATSC environment, this field represents the major channel number. In the DVB environment, the original network id occupies bits 31 - 16, and the transport stream id occupies bits 15-0.
`minor_chan_num`	In the ATSC environment, this field represents the minor channel number. In the DVB environment, this field is set by the Channel Manager to the service id of the channel.
`chan_user`	may be defined by applications. The `nav_chan_set_userdef()` API call can alter the contents of this field. The Channel Manager does not interpret this field.

`chan_flags`	This field is a bit mask. If bit 0 is set, that indicates that the channel is deleted. If bit 1 is set, then it indicates that the channel is “hidden” in that it contains only application data and does not carry MPEG audio and video. The other bits are currently unused.
`chan_nbr_fav_rings`	is the number of favorite channel rings in which this channel appears.
`chan_rsvd`	is reserved for future use.

ATSC and DVB specific channel information

As explained before, each `channel` structure contains a pointer to a `chan_info` structure that contains detailed information about a channel. This `chan_info` structure is actually the first field of another structure that contains additional fields which are specific to the protocol (ATSC or DVB) being used. Figure 2-3 illustrates this for the ATSC standard. The enclosing structure is the `atsc_chan_info` structure. A similar structure `dvb_chan_info` is defined for DVB.

Figure 2-3 Protocol Specific Channel Information



Applications can access the ATSC or DVB specific information using the `ch_info` pointer in the channel structure. However, doing so will restrict portability of the application to the specific protocol in use.

The `atsc_chan_info` structure is defined in `<SPF/atscpsip.h>` as follows:

```
typedef struct _atsc_chan_infoatsc_chan_info;
typedef struct _atsc_chan_info*Atsc_chan_info;

typedef struct _atsc_info
{
  chan_info      chan_info_common;
  Atsc_freq_info atsc_freq_info_ptr;
  Atsc_chan_info atsc_next_chan_on_freq;
  Atsc_chan_info atsc_prev_chan_on_freq;
  u_int32        atsc_delete_time_stamp;
  u_int16        atsc_short_name[7];
  u_int16        atsc_trnsprt_strm_id;
  u_int16        atsc_pgm_nbr;
  u_int16        atsc_service_info;
  u_int16        atsc_source_id;
  u_int16        atsc_pcr_pid;
  u_int16        atsc_video_pid;
  u_int16        atsc_audio_pid;
  u_int8         atsc_video_stream_type;
  u_int8         atsc_audio_stream_type;
  u_int8         atsc_modulation_mode;
  u_int8         atsc_chan_flags;
};
```

The `atsc_short_name`, `atsc_trnsprt_strm_id`, `atsc_pgm_nbr`, `atsc_service_info`, and `atsc_source_id` fields are copied directly from the ATSC Terrestrial Virtual Channel Table, The remaining fields are for internal use by the Channel Manager Protocol Driver.

The dvb_chan_info structure is defined in the header file <SPF/dvb.h> as follows:

```
typedef struct _dvb_chan_info      dvb_chan_info;

typedef struct _dvb_chan_info      *Dvb_chan_info;
```

```
struct _dvb_chan_info
{
chan_info        chan_info_common;
u_int16          dvb_service_onid;
u_int16          dvb_service_tsid;
u_int16          dvb_service_id;
u_int8           dvb_service_flags;
u_int8           dvb_service_type;
char   dvb_service_name[DVB_MAX_NAME_SIZE];
char   dvb_service_provider_name[DVB_MAX_NAME_SIZE];
dvb_link         dvb_service_links[4];
u_int8           *dvb_service_descriptors;
u_int32          dvb_service_desc_size;
Dvb_freq_info    dvb_freq_info_ptr;
Dvb_chan_info    dvb_next_chan_on_freq;
Dvb_chan_info    dvb_prev_chan_on_freq;
u_int32          dvb_delete_time_stamp;
u_int8           dvb_eit_version;
u_int8           dvb_reserved[3];
};
```

The `dvb_service_onid`, `dvb_service_tsid` and `dvb_service_id` represent the represent the original network id, transport id and service id of the channel.

The `dvb_service_name` and `dvb_service_provider_name` contain text with the name of the channel and the name of the channel provider.

The `dvb_service_flags` field is interpreted as follows

```
Bit  7: EIT schedule table available for channel
Bit  6: EIT presentfollowing table available
```

```
Bit 3-5: Running Status of channel
Bit 2: Free CA Mode ( No Conditional Access)
Bit 1: Unavailable in current country
Bit 0: Channel information is being updated
```

The remaining fields are use by the Channel Manager driver only.

Other Structures

Another structure used by the Navigation API is the nav_notify structure. This structure is described in detail on the following page.

nav_notify

Declaration

The `nav_notify` structure is declared in the file `SPF/nav_api.h` as follows:

```
typedef struct nav_notify
{
   u_int32type;
   u_int32value;
   u_int32ev_id;
} nav_notify, *Nav_notify;
```

Description

The `nav_notify` structure is used to specify the desired notification method for asynchronous requests to the Navigation API.

Fields

`type`	defines the type of notification. This field should be set to either `NAV_NTFY_SIGNAL` or `NAV_NTFY_EVENT`.
`value`	is set to the value of the signal or event.
`ev_id`	is set to the event id if the notify is by event.

Using the Navigation API

Accessing the API

To access the Navigation API, an application must include the header file `SPF/nav_api.h` and link to the `nav_api.l` library.

Initializing

Each process which uses the Navigation API must issue the `nav_init()` call as its first call to API. In response to this call, the API will open a path the the Channel Manager Protocol Driver on behalf of the calling process. The Navigation API will use this path to pass subsequent API requests to the Channel Manager.

The `nav_init()` call requires a pointer to a `boolean` data type as a parameter. This boolean data type is used to specify whether or not the Channel Manager Protocol Driver should automatically save the channel map data structures to non-volatile memory when the last process using the API issues the `nav_term()` call. If the Navigation API is initialized so that the Channel Manager does not automatically save the channel map to non-volatile memory, then the last process to terminate use of the API should copy the channel map to some non-volatile memory that it manages before calling `nav_term()`. Upon a subsequent re-boot, the application can pass the channel map back to the Channel Manager Protocol Driver by issuing the `nav_set_chan_map()` call after calling `nav_init()`.

If the Channel Manager has saved the channel map data structures to non-volatile memory, then the application can request the Channel Manager to re-load the channel map following a subsequent re-boot by issuing the `nav_load_chan_map()` call after calling `nav_init()`.

Building the Initial Channel Map

On the very first boot sequence for a new receiver, a copy of the channel map does not yet exist in non-volatile memory. A frequency scan procedure is necessary to build this initial channel map. The frequency scan procedure tests each frequency which has been reserved for television reception for the presence of either an analog or digital signal. If a signal is detected on a given frequency, the application performing the frequency scan should add that frequency to the channel map.

To perform a frequency scan procedure, the application should execute the following steps for each possible frequency:

1. Issue the `nav_tune()` call to tune to the frequency to be tested. The `nav_tune()` call requires a pointer to a structure which specifies the frequency value and the desired tuner (i.e., analog or digital).
2. Issue the `nav_get_signal_info()` call to measure the signal strength on the tuned frequency.
3. If a signal is detected, add the frequency to the channel map by calling the `nav_add_digital_freq()` or `nav_create_analog_chan()` function. The `nav_add_digital_freq()` call should be issued if the digital tuner was specified in the `nav_tune()` call, and the `nav_create_analog_chan()` function should be issued if the analog tuner was specified.

Note

For the ATSC Channel Manager, digital frequencies passed to the `nav_tune()` call should be 310 KHz above the lower edge of the frequency's RF band. Similarly, for the ATSC Channel Manager, analog frequencies passed to the `nav_tune()` call should be 1.25 MHz above the lower edge of the frequency's RF band.

4. If more frequencies remain to be tested, return to Step 1.

On boot-up, an application can determine whether a channel map is present in non-volatile memory by issuing the `nav_get_configuration()` call. If no channel map is present, the application can perform the frequency scan procedure.

A frequency scan procedure may also need to be re-executed if the receiver is moved from one geographic area to antoher area. For this case, the `nav_delete_chan_map_file()` call can be issued to delete the old channel map before the new one is created.

Channel Change Requests

Channel change requests to the Navigation API may return an error code as the status of the call. For example, an error will be returned if an invalid channel number is specified or if the Tuner Device Driver returns an error when attempting to switch to the channel's frequency.

An error code of `EOS_CH_NOT_ACTIVE` will be returned if the Channel Manager was able o tune to the proper frequency, but the specified channel was not currently active. In this case, the Channel Manager will remember the requested channel number as though it is the active channel. Subsequent calls to change to the next or preceeding channel will use this remembered channel number as the starting point for selecting the next or preceeding channel for playout.

If the `EOS_CH_NOT_ACTIVE` error status is returned, the application can also request to receive an asynchronous notification if the channel map for the channel's frequency changes. When this notification is received the application can re-attempt to channel change request in order to determine if the requested channel has become active. This allows a viewer to tune to a desired channel number several minutes before the channel becomes active, with the appearance that the channel will automatically start playing once it does become active.

Accessing the Channel Map Data Structures

Although function calls are provided by the Navigation API for retreiving information from the channel map, an application may also directly read the channel map's data structures. For such cases, an application may require that the channel map remain in a consistent state while multiple fields in the channel map are examined. In order to accomplish this, an application should issue the `nav_lock_map()` call to lock the channel map from updates before examining the first field. Similarly, the `nav_unlock_map()` call should be issued to unlock the channel map after examining the last field. While the channel map is locked, no changes to it due to incoming SI data or Navigation API calls from another process will be allowed.

Setting User Preferences

In some cases an MPEG program contains several audio elementary streams, each associated with a different language. When a channel change operation occurs for such cases, the Real-Time Device Driver selects the audio stream to be decoded based on a set of user preference settings maintained by the driver. The Navigation API call `nav_get_preferences()` can be used to read these settings, and the `nav_set_preferences()` call can be used to set them. For these calls, the application must pass a pointer to a `ch_user_pref` structure as a parameter in the call.

For More Information

Refer to *Chapter 6: The Channel Manager Protocol Driver* for information about the `ch_user_pref` structure.

When selecting an audio stream for a program, an attempt is first made to find an audio stream matching the primary language preference. If one is not found an attempt is made to find an audio stream matching the secondary language preference. If this also fails, the first audio stream specified in the Program Map Table is used.

Note

If a program is being decoded when a `nav_set_preferences()` call is made, an audio stream is re-selected based on the new settings.

WARNING

The current preference settings are lost when power is turned off or the system is re-booted. Application programs should save the preference settings to non-volatile memory when they change and issue a `nav_set_preferences()` call with these settings anytime the system has been re-booted.

Requesting Notifications

The Navigation API calls to change the current channel and change the current channel ring can be specified to execute either synchronously or asynchronously. When specified to execute asynchronously, these calls return control to the calling application before the requested operation has completed. In this case the calling application can specify a method by which to be notified when the operation has completed. This notification can be via either a signal or an event.

An application specifies the execution mode (synchronous or asynchronous) and the notification method for asynchronous calls through the `ntfy` parameter. The `ntfy` parameter is a pointer to a `nav_notify` structure owned by the application. If this pointer is NULL, a synchronous call is executed. Conversely, if this pointer is not NULL, an asynchronous call is executed. In this case the `ntfy` parameter points to a `nav_notify` structure specifying the desired notification method.

The contents of the `nav_notify` structure are copied during an asynchronous Navigation API call before the call returns control to the calling application. Thus, a calling application may de-allocate or reuse the structure before the asynchronous call has completed.

The asynchronous Navigation API calls also require a pointer to a `nav_status` structure as a parameter. The `nav_status` structure simply provides a location for an error code to be returned. Thus, this structure must remain allocated and unused by the application until the asynchronous call has completed. When an application is notified (either by a signal or an event) that an asynchronous call has completed, it should check the `nav_status` structure to determine whether or not the call was successful.

Note

Asynchronous calls perform some error checking on the passed parameters immediately, but other error conditions can not be detected until the requested operation has been attempted. Thus, some error codes are returned immediately as the return value of the call itself, while other error codes are returned in the `nav_status` structure. When the return value from the asynchronous call is anything other than `SUCCESS`, an immediate error was detected and the return value is the error code. In this case, the requested operation was not issued and therefore the asynchronous notification never occurs. When the return value from the asynchronous call is `SUCCESS`, the asynchronous operation has been issued. However, this does not imply the operation necessarily completes successfully. In this case, the requested asynchronous notification occurs and the actual status of the operation is returned in the `nav_status` structure.

WARNING

If the nav_status structure is released before the asynchronous call is completed either because it was allocated on the stack or because the application freed it, then the Channel Manager driver will be writing into free memory which can cause a lot of problems in the system.

Terminating

Applications using the Navigation API must call nav_term() before exiting. The Navigation API may allocate memory on behalf of the application, and this memory is freed when the nav_term() call is processed. If this is the last application using the Navigation API, then the Channel Manager driver will store the channel information to NVRAM. Typically if a user powers down the SetTop Box using the Power Key on the remote control device, the application should trap that key and call nav_term() before powering down the SetTop Box. This will give the Channel manager a chance to save information on NVRAM.

The EPG API

Purpose of the API

The EPG API supports an EPG application by allowing it to retrieve information from the Channel Manager driver about events playing on specific channels at a specific time. The API presents the application with an easy to use interface for obtaining information about available events on the network, organizing the events into a configurable cache, and managing the memory associated with the cache.

Architecture of the API

The EPG API is implemented as a set of function calls whose object code is contained in a statically-linked library named `nav_api.l`. Applications issuing calls to the EPG API must be linked to this library. The EPG API is an extension to the Navigation API which is also included in `nav_api.l`

The architecture of the EPG API is built around the event cache and the event data structures.

The event data structure encapsulates one TV programming event on a specific channel from a specific start time to a specific end time. Apart from the channel, start time and duration, the event structure also contains text describing the title and the description of the event.

An event cache is a collection of events retrieved from the SI tables on the network. The event cache data structure has been created to closely match the requirements of the EPG Grid based User Interface paradigm. In this paradigm, the EPG application displays the programming schedule using a two dimensional grid format. Each row of the grid displays the program schedule for one channel, while each column represents a time-slot of typically a half-hour duration. Using the EPG API the EPG application can create one or more event cache structures and request the Channel Manager to fill them with schedule information. Each event cache can hold a configurable number of events based on channel and time.

EPG Data Structures

The primary data structure maintained by the Channel Manager Protocol Driver but made available to application program is the `epg_event` structure. Also of significance is the `epg_cache_params` structure which is used by the application to specify EPG cache parameters when it is being created. These structures are defined in the file `<SPF/epg_api.h>` and further defined below.

epg_event

Declaration

The `epg_event` structure is declared in the file `SPF/epg_api.h` as follows:

```
typedef struct _epg_event       epg_event;
typedef struct _epg_event       *Epg_event
struct _epg_event
{
    Epg_event       ev_next;
    Epg_event       ev_prev;
    time_t          ev_start_time;
    u_int32         ev_duration;
    u_int32         ev_ring_id;
    u_int32         ev_major_channel_num;
    u_int32         ev_minor_channel_num;
    u_int16         ev_id;
    u_int8          ev_flags;
    u_int8          ev_reserved;

};
```

Description

The `epg_event` structure contains the event information fields common to both DVB and ATSC environments.

Fields

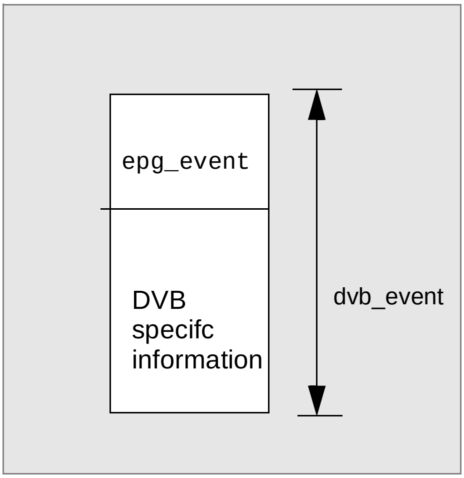
`ev_next`	Used to link to the next event in the cache.
`ev_prev`	Used to link to the previous event in the cache.
`ev_start_time`	Start time of the event in UTC.

`ev_duration`	Duration of the event in seconds.
`ev_ring_id`	Id of the ring containing the channel to which the event belongs
`ev_major_chan_num`	Major Channel Number of the channel to which the event belongs. For DVB, this conatins the original network id in 16 MSB and the transport id in the 16 LSB.
`ev_minor_chan_num`	Minor Channel Number of the channel to which the event belongs. For DVB, this is the same as the service id of the event.
`ev_id`	Event id.
`ev_flags`	Bit mask of flags. Currently the only bit used is bit 0. If this is set to 1 then the event is assumed to be deleted and invalid.

ATSC and DVB specific event information

Similar to the channel information structure, the epg_info structure is also the first field of another structure that contains additional fields which are specific to the protocol (ATSC or DVB) being used. Figure 2-3 illustrates this for the DVB standard. The enclosing structure is the `dvb_event` structure. A similar structure `atsc_event` is defined for ATSC.

Figure 2-4 Protocol Specific Event Information



Declaration

The `dvb_event` structure is declared in the file `SPF/dvb.h` as follows:

```
typedef struct _dvb_event       dvb_event;
typedef struct _dvb_event       *Dvb_event
struct _dvb_event
{

epg_event        ev_common;
u_int8           dvb_flags;
u_int8           dvb_rating;
u_int8           dvb_name_length;
u_int8           dvb_text_length;
char             *dvb_name;
char             *dvb_text;
Dvb_component    dvb_streams;
Dvb_link         dvb_links;
Dvb_content      dvb_content_types;
u_int8           dvb_num_of_streams;
u_int8           dvb_num_of_links;
u_int8           dvb_num_of_content_types;
u_int8           dvb_reserved;
u_int8           *dvb_descriptors;
u_int16          dvb_desc_size;
u_int16          dvb_extended_text_length;
char             *dvb_extended_text;

};
```

Description

The dvb_event structure contains the event information fields for DVB environments.

Fields

ev_common	The part of the event structure common to both DVB and ATSC.
dvb_flags	Bit flag. Bit 5-7 contain the running status of the event. Bit 4 is set if the event has no Conditional Access associated.
dvb_rating	If non-zero, then it it is the minimum age needed to view the event - 3. If zero, then the field is invalid.
dvb_name_length	length in bytes of the name field.
dvb_text_length	length in bytes of the event description field.
dvb_name	Pointer to name or title of the event.
dvb_text	Pointer to textual desciption of the event.
dvb_streams	Array of component audio/video stream structures. The dvb_component structure is specified in <SPF/dvb.h>.
dvb_links	Array of links to other channels (refer DVB-SI specification). The dvb_link structure is specified in <SPF/dvb.h>
dvb_content_types	Array of structures describing the content type of the event. The dvb_content structure is specified in <SPF/dvb.h>
dvb_num_of_streams	Number of elements in the dvb_streams array.
dvb_num_of_links	Number of elements in the dvb_links array.
dvb_num_of_content_types	Number of elements in the dvb_content_types array.

`dvb_descriptors`	Pointer to the event descriptor buffer.This includes descriptors from the EIT table that are cached as per the device descriptor for the channel manager.
`dvb_desc_size`	Size of the event descriptor buffer.
`dvb_extended_text_length`	Size in bytes of the extended text description of the event
`dvb_extended_text`	Extended text describing the event.

atsc_event

Declaration

The `atsc_event` structure is declared in the file `SPF/atscpsip.h` as follows:

```
struct _atsc_event
{
    epg_event       ev_common;
    u_int8          atsc_name_length;
    u_int8          atsc_text_length;
    u_int8          atsc_reserved[2];
    char            *atsc_name;
    char            *atsc_text;
};
;
```

Description

The `atsc_event` structure contains the event information fields for ATSC environments.

Fields

`ev_common`	The part of the event structure common to both DVB and ATSC.
`atsc_name_length`	length in bytes of the name field.
`atsc_text_length`	length in bytes of the event description field.
`atsc_name`	Pointer to name or title of the event.
`atsc_text`	Pointer to textual desciption of the event.

Declaration

The `epg_cache_params` structure is declared in the file `SPF/epg.h` as follows:

```
typedef struct _epg_cache_params
{
int         cache_type;
u_int32     cache_size;
u_int32     max_num_of_events;
u_int32     max_event_name_size;
u_int32     max_event_short_text_size;
u_int32     max_event_long_text_size;

} epg_cache_params, *Epg_cache_params;
```

Description

The `epg_cache_params` structure is used by application to specify the type and size of an EPG cache during cache creation. The parameters listed in the cache parameters structure is used to allocate memory for the cache.

The cache_type parameter can be either `CACHE_TYPE_VARIABLE_EVENT_SIZE` or `CACHE_TYPE_FIXED_EVENT_SIZE`. When this parameter is passed as `CACHE_TYPE_VARIABLE_EVENT_SIZE`, the cache_size field should also be passed. In this case, a cache will be created with this specified size (in bytes). When such a variable event size cache is filled (with the `nav_add_to_epg_cache()` call) the event information placed into the cache will not be truncated. However, since the size of each event's title and text description may vary, the number of events that can be placed into a variable event size cache is unpredictable. Thus, subsequent calls to `nav_add_to_epg_cache()` for a variable event size cache may fail to return all the events if the requested information does not fit into the cache.

When the `cache_type` field is specified as `CACHE_TYPE_FIXED_EVENT_SIZE`, the limits on the size fields should also be passed. In this case, a cache will be created to hold the specified number of events, with the specified amount of memory allocated for each event's title and text description. If the actual size of an event's title or text description is bigger than the specified size for a subsequently requested event, the title or text description will be truncated to fit.

Thus, by specifying the `cache_type` field, applications can choose between the following two options:

1. a cache that is guaranteed to hold the specified number of events, but for which the event title or text description may be truncated, and

2. a cache that can hold an undetermined number of events, but for which the event title and / or text description are never truncated.

EPG applications which only display an event's title and / or text description, but do not process the information in any other way, may require an event's title or text description to be truncated to fit into a fixed size display area on the screen. Such applications can use fixed event size EPG caches with the event title and description size specified to match the sizes that can be displayed. This will simplify such applications by eliminating problems that can occur when the number of events requested during an nav_add_to_epg_cache() may otherwise overflow a variable event size EPG cache.

Fields

`cache_type`	This field can take the following values: CACHE_TYPE_VARIABLE_EVENT_SIZE and CACHE_TYPE_FIXED_EVENT_SIZE. In the former type, the size of an individual event can vary and the application only specifies the total size of the cache. Such a cache is fixed in size but can contain a variable number of events. In the latter type, the size of an individual event is fixed and is specified by listing maximum lengths for the event name, description and long

description text. Such a cache can contain only a fixed specified number of events of a specified size.

`cache_size`	Total size in bytes of the cache. Used only for CACHE_TYPE_VARIABLE_EVENT_SIZE.
`max_num_of_events`	Maximum number of events in the cache. Used only for CACHE_TYPE_FIXED_EVENT_SIZE.
`max_event_name_size`	Maximum size in bytes of the title of the event. Used only CACHE_TYPE_FIXED_EVENT_SIZE.
`max_event_short_text_size`	Maximum size in bytes of the short description of the event. Used only CACHE_TYPE_FIXED_EVENT_SIZE.
`max_event_long_text_size`	Maximum size in bytes of the long description of the event. Used only CACHE_TYPE_FIXED_EVENT_SIZE.

Using the EPG API

Accessing the API

To access the EPG API, an application must include the header file `SPF/epg_api.h` and link to the `nav_api.l` library.

Initializing

Since the EPG API is part of the Navigation API, the application must issue the `nav_init()` call as its first call to API. In response to this call, the API will open a path the the Channel Manager Protocol Driver on behalf of the calling process. The API will use this path to pass subsequent API requests to the Channel Manager.

Before using the calls from the EPG API, the application must ensure that the channel map has been built or loaded from NVRAM.

Retrieving Current Event Information

One of the typical requirements of an EPG application is to displayed information about the currently playing event on the currently viewed channel. This is usually displayed during a channel change operation or when the INFO button on the remote control is pressed.

The current event information is automatically cached and constantly updated by the Channel Manager driver. For an application to get this information and to keep track of the changes to it, it must execute the following steps.

1. Issue the `nav_notify_asgn()` call with the notification type set to `NAV_NTFY_ON_CURR_EVENT_AVAILABLE.` This call sets up a notification to be received from the Channel Manager driver when new current event information is available. This request is persistent and thus only needs to be made once.

2. When this call is made, if the information about the current event is available, then the notification is sent right away. If not, the Channel Manager driver sends the notification when current event information becomes available from the network SI tables.
3. When the application receives the notification, it must issue the call `nav_get_curr_event()` to retrieve the current event information. Note that the driver returns a pointer to a copy of the current event information and not its own internal event structure.
4. Subsequently, when the current event information changes either because the channel was changed, or a new event started on the current timeslot, the driver sends the notification to the application. The application must then reissue the `nav_get_curr_event()` call.

Retrieving EPG schedules

EPG schedule information is managed via the event cache structure. To collect EPG schedule information the application must execute the following steps:

1. Create an EPG cache using the `nav_create_epg_cache()` call. The size and type of the cache are allocated as per memory availability and EPG grid usage. For e.g., if a fixed number of EPG grid entries are to be refreshed on the screen at a time, then it is best to allocate a cache of type `CACHE_SIZE_FIXED_EVENT_SIZE` and specify the maximum number of entries in the cache in the `epg_cache_params` structure.
2. Fill the EPG cache using the asynchronous call `nav_add_to_epg_cache()`. Note that this call actually waits until the information is retrieved from the SI tables on the network and store in the cache. If for some reason this call needs to be terminated prematurely, then the `nav_cancel_epg_retrieval_request()` may be used.

3. Once the cache is filled, the individual events can be obtained from the cache using the call `nav_get_event_info().` This call does not get any information from the network. It simply selects events alreayd present in the cache based on channel and time criteria set by the application
4. Subsequently, events may either be deleted from the cache using the `nav_remove_from_epg_cache()` call in order to allow more events to be added to the cache.
5. Or the entire cache may be detroyed using the `nav_destroy_cache()` call.

Retrieving Channel Lists

In order to draw an EPG Grid Screen, an EPG application must first determine the list of channels that are to be displayed in the grid. When a grid-based EPG application is first launched, the application typically displays a list of channels with the current channel in the middle of the grid. For example, consider an EPG grid that can display scheduling information for seven channels simultaneously. If the EPG application is launched when the currently tuned channel number is 7-1, the EPG application will display the grid with channel 7-1 in the middle row of the grid, the three preceding channels above this row, and the three succeeding channels below this row.

Thus, an EPG application must be able to determine the currently active channel as well as the subset of channel numbers that immediately precede this channel and the subset of channel numbers that immediately succeed this channel. The following calls provided by the Navigation API can be used for this purpose:

- `nav_get_cur_chan_num():`

Retrieves the current channel's channel number.

- `nav_list_preceding_major_channels():`

Retrieves the list of major channel numbers that immediately precede a specified major channel number.

- `nav_list_succeeding_major_channels():`

Retrieves the list of major channel numbers that immediately succeed a specified major channel number.

- `nav_list_preceding_minor_channels():`

Retrieves the list of minor channel numbers on a specified major channel number that immediately precede a specified minor channel number.

- `nav_list_succeeding_minor_channels():`

Retrieves the list of minor channel numbers on a specified major channel number that immediately succeed a specified minor channel number.

- `nav_update_minor_channel_list():`

Force the Channel Manager driver to update its list of minor channel numbers for a specific major channel number. This is required for e.g. in terrestrial broadcast environments where the SI information on frequency does not include SI information about other frequencies. This means that when the Channel Manager is tuned to one frequency, it could have stale SI information about other frequencies.

Getting Time from the network

The EPG API provides the `nav_get_network_time()` call to get the time specified (in UTC format) in the SI tables. Note that this time is not always guaranteed to be available and updated, since the user could tune to an analog channel on which no SI data is present.

The MPEG Private Data API

Purpose of the API

The Mpeg Private Data API allows Electronic Program Guide (EPG applications as well as other types of applications) to obtain information from the incoming MPEG-2 transport stream. Although primarily intended for EPG applications, the API is robust enough to support any application wishing to retrieve any information conforming to the MPEG-2 section syntax from the MPEG-2 bit stream.

This section describes the functionality and the calls available for applications in the Mpeg Private Data API.

In general it is assumed EPG information is carried in MPEG-2 transport streams using Private PSI sections. (Refer to ***ISO/IEC 13818-1.***) There are two major standard bodies which have defined standards for carrying EPG data in MPEG-2 multiplexes. The DVB Project defines a DVB-SI standard (***ETS 300 468***) and the ATSC defines another specification for Digital Television (***Doc A/65***). The two specifications are quite different in the nature of the information specified. However the approach is the same in both cases. Both standards define a set of private tables carrying information descriptors. The tables are comprised of sections with each section complying to the Private Section syntax defined in ISO/IEC 13818-1. In addition, some EPG vendors have added extensions to the DVB and ATSC standards allowing additional information to be carried in the MPEG-2 stream. Such extensions either define additional tables or define additional descriptors appearing in the existing tables.

For More Information

For more information about these standards, contact the following sources:

ISO/IEC 13818-1 may be obtained from the intenet website http://www.iec.ch/

ETS 300 468 may be obtained from The Publications Office, ETSI, 06921 Sophia Antipolis CEDEX, France

Doc A/65 may be obtained from the internet website http://www.atsc.org

Keeping in mind the differences in content between the different standards and the proprietary nature of some of the EPG information, this Mpeg Private Data API does not attempt to parse the tables in the system layer and return parsed information to the application. What it provides instead is strong support for obtaining tables and their component sections to applications. The calls provided can be used to obtain tables or specific sections within the tables meeting a certain criteria.

Architecture of the API

Typically an application registers with the Mpeg Private Data API for a certain table (or a set of sections within a table) from the incoming MPEG-2 transport stream. In response, the API issues the appropriate system call to the SPF Real-Time Network Driver. The requested sections are read from the network into standard networking mbufs and passed up to the application layer. Typically, each mbuf contains one section; however, when the Program Association Table is requested, all of the sections are concatenated into a single mbuf.

The application can either request the mbuf itself or can request the section be copied from the mbuf into its own buffer. Getting the mbuf itself allows the application to cache just the information it needs and a copy operation between system memory and user memory. When the application has the complete table, it can either unregister for that table with the API or change the mask used to select sections so only sections with the next version number are selected. Note if it does not do either then the sections continue to be cached at the system layer as they arrive since the tables are repeated periodically in a broadcast environment. This may cause the system's mbuf pool to be depleted. The application can also register for a notification when section data is available.

Even though this API can be used to get any table, some tables are used by the system layer. Specifically the Program Map Table and Program Association Table are used by the Real-Time Device Driver. Also the Channel Manager Protocol Driver uses the Network Tables carrying delivery information and the service/program listing. This API supports multiple requests for the same table, but this mode of operation is slightly slower and uses more memory. Hence it is recommended that applications using the API request only the tables carrying program guide information if possible.

Using the API

Accessing the API

To access the Mpeg Private Data API, an application must include the header file `SPF/mpg_api.h` and link to the `mpg_api.l`, `item.l`, and `mbuflib.l` libraries.

Terminating

Applications using the Mpeg Private Data API must call `mpg_term()` before exiting. The Mpeg Private Data API may allocate memory and cache mbufs on behalf of the application, and this memory is freed when the `mpg_term()` call is processed.

The Parental Control API

Purpose of the API

The Parental Control API allows the end-user to restrict Television viewing by members of the household. The API provides multiple users with access to the Set-top Box via password protected user accounts. User accounts may be classified as "parent accounts" and "child accounts". Parent accounts can regulate what TV Programs (events) are accessible to the child accounts. Access may be restricted based on

- Time of the day. For e.g. a child may be allowed to watch TV only for 2 hours in the evening.
- Rating of the event. For DVB, a minimum age is specified for each event. For ATSC, rating could be provided on multiple dimensions. For e.g. the most popular rating dimension in the United States is the MPAA rating scheme.
- Channel number. For e.g. a child may not be allowed to view anything but the Disney channel.

All account information is stored on some non-volatile storage and is hence available between power-cycles to the box.

This API will be used by an application which is responsible for providing the User-Interface needed for the Parental Control functions. The API is not responsible for the User-Interface functionality. The section below describes how a typical parental control application would use this API. Note that we are not implying the existence of an independent application. The functionality of this application can and probably should be absorbed into the EPG or Player Shell application.

Using the API

Accessing the API

To access the Parental Control API, an application must include the header file `SPF/pc.h` and link to the `pc_api.l` library.

Parental Control Application

This section explains how a typical parental control application uses the API.

The Set-top Box will be shipped with one factory installed Parent account with a suitable default password. When the user first turns on the Set-top Box he/she is allowed access to the pre-installed account via a configuration menu. The specification of the configuration menu depends on the OEM and is beyond the scope of this document. Using the parent account the user can set up different accounts for different members of the household. Each account can have specific privileges, which may be changed by any parent account at any time. A guest or default account may also be created which is used if a user login fails. The user privileges are stored by the API on some non-volatile media and are hence persistent between power cycles.

When the Set-top Box is turned on, the Parental Control application prompts the user for a login. If the user hits cancel, then the default or guest account is activated, if one exists. Otherwise all access is restricted and the user cannot watch any TV programs. If the user logs in correctly, then he/she can watch TV content as specified in his/her account privileges. If another user logs in then his/her account privileges override the previous user's login, which is to say that the Set-top Box simply uses the privileges of the last user that logged in. When the current user logs out, the system switches to the Guest or default account.

At any point in time, the Parent account can choose to disable Parental Control. This keeps all the account information intact, but just disables the Parental Control functionality. Users are not shown user login screens when Parental Control is disabled. When Parental Control is re-enabled, the user accounts are re-activated and don't have to be created again.

Parental Control Data Structures

The primary data structures maintained by the Channel Manager Protocol Driver but made available to application program via the Parental Control API are the `pc_account` and the `pc_privilege` structures. These structures are defined in the file `<SPF/pc.h>` and further defined below.

pc_account

Declaration

The `pc_account` structure is declared in the file `SPF/pc.h` as follows:

```
typedef struct _pc_account pc_account;
struct _pc_account
{
char                login_name[PC_MAX_LOGIN_NAME_SIZE];
char                password[PC_MAX_PASSWORD_SIZE];
boolean             is_parent;
int                 default_channel_privilege;
int                 default_time_privilege;
pc_privilege*       privilege_list;
pc_account*         next;
};
```

Description

The `pc_account` structure contains information about a user account.

Fields

`login_name`	Login name of the account.
`password`	Password for the account.
`is_parent`	If set to TRUE, then this account is a parent account
`default_channel_privilege`	If zero, then by default all channels are offlimits unless explicitly allowed via a channel privilege. If non-zero, then all channels are viewable unless explicitly restricted via a channel privilege.

`default_time_privilege`	If zero, then by default the TV is offlimits at all times of the day, unless explicitly allowed via a time privilege. If non-zero, then all channels are viewable at all times of the day unless explicitly restricted via a time privilege.
`privilege_list`	List of privileges on account.
`next`	Used to link the account into a link list of accounts in the driver. For driver use only.

pc_privilege

Declaration

The pc_privilege structure is declared in the file SPF/pc.h as follows:

```
typedef struct _pc_privilege pc_privilege;
struct _pc_privilege
{
u_char     privilege_type;
u_char     viewer_age;
u_char     rating_dimension_index;
u_char     rating_value_index;
u_int32    major_channel_number;
u_int32    minor_channel_number;
u_char     start_hour;
u_char     start_minute;
u_int16    duration;
pc_privilege* next;
};
```

Description

The pc_privilege structure contains information about a privilege granted on a user account.

Fields

privilege_type	Type of privilege structure. This could be PC_CHANNEL_PRIVILEGE, PC_TIME_PRIVILEGE, PC_AGE_PRIVILEGE and PC_RATING_PRIVILEGE.

viewer_age	The age of the current user. This is used for DVB only with privilege_type set to PC_AGE_PRIVILEGE
rating_dimension_index	This is used for ATSC only with privilege_type set to PC_RATING_PRIVILEGE. It indicates which rating dimension index from the RRT table is currently selected.
rating_value_index	This is used for ATSC only with privilege_type set to PC_RATING_PRIVILEGE. It indicates which rating value index from the RRT table is currently selected.
major_channel_number	This is used if the privilege_type is set to PC_CHANNEL_PRIVILEGE. It interpretation depends on the default channel rating of the current user. For e.g. if the default channel rating is 0, then the user is allowed to view the channel specified by major_channel_number and minor_channel_number.
minor_channel_number	This is used if the privilege_type is set to PC_CHANNEL_PRIVILEGE. It interpretation depends on the default channel rating of the current user. For e.g. if the default channel rating is 0, then the user is allowed to view the channel specified by major_channel_number and minor_channel_number.
start_hour	This is used if the privilege_type is set to PC_TIME_PRIVILEGE. It interpretation depends on the default time rating of the current user. For e.g. if the default time rating is 0, then the user is allowed to view the TV from the specified start time to the

	specified end time. This value represents the hour after midnight of the specified start time.
`start_minute`	This is used if the privilege_type is set to `PC_TIME_PRIVILEGE`. It interpretation depends on the default time rating of the current user. For e.g. if the default time rating is 0, then the user is allowed to view the TV from the specified start time to the specified end time. This value represents the minute after the hour of the specified start time.
`duration`	This is used if the privilege_type is set to `PC_TIME_PRIVILEGE`. It interpretation depends on the default time rating of the current user. For e.g. if the default time rating is 0, then the user is allowed to view the TV from the specified start time to the specified end time. This value represents the duration in minutes of the time privilege.
`next`	Used to link the account into a link list of privileges for the account in the driver. For driver use only.

Chapter 3: The Navigation API

This chapter contains descriptions, in alphabetical order, of Navigation Management functions.

Function Descriptions

The function descriptions are, for the most part, self-explanatory. Each function description contains the following sections:

The SYNTAX section shows the function prototype with the required parameters and their data types.

The DESCRIPTION section provides a description of the function.

The PARAMETERS section provides details about each of the function's parameters.

FATAL ERRORS are errors detected within the API call and are returned directly by that particular call. Applications may not be able to recover from fatal errors.

API specific Errors are errors detected within the API call and are a direct result of that particular call. Applications can recover from API specific Errors.

INDIRECT ERRORS are errors returned by another function called during the processing of the API request.

The SEE ALSO section lists related functions or materials that provide more information about the function.

Navigation Management Functions

The table below summarizes the Navigation Management functions:

Table 3-1 Summary of Navigation Management Functions

Function	Description
`nav_abort()`	Stop decoding and playout on the current channel.
`nav_add_chan()`	Add a channel to a favorite channel ring.
`nav_add_digital_freq()`	Add a digital frequency to the channel map.
`nav_add_to_epg_cache()`	Add requested events to an EPG grid cache.
`nav_cancel_epg_retrieval_request()`	Cancel an EPG add to cache request.
`nav_chan_set_userdef()`	Set the user-defined field of a chan_info structure.
`nav_clear_chan_map()`	Reset the channel map to an empty state.
`nav_create_analog_chan()`	Create an analog channel in the channel map.
`nav_create_chan_map()`	Create an empty channel map.
`nav_create_epg_cache()`	Create an empty EPG grid cache.

Table 3-1 Summary of Navigation Management Functions (continued)

Function	Description
nav_create_ring()	Create a new favorite channel ring.
nav_delete_chan_map_file()	Delete the channel map file from the NRF file system.
nav_destroy_analog_chan()	Remove an analog channel from the channel map.
nav_destroy_epg_cache()	Destroy an EPG grid cache.
nav_destroy_ring()	Destroy a favorite channel ring.
nav_flush_epg_cache()	Flush all events from the specified EPG grid cache.
nav_get_chan()	Retrieve a pointer a channel's `channel` structure.
nav_get_configuration()	Retrieve the Channel Manager's configuration settings.
nav_get_cur_chan()	Retrieve a pointer to the current channel's `channel` structure.
nav_get_cur_chan_num()	Get the major/minor channel number for the current channel.
nav_get_cur_ring()	Retrieve a pointer to the current channel ring's `ring` structure.
nav_get_current_event()	Get the current event on the current channel.

Table 3-1 Summary of Navigation Management Functions (continued)

Function	Description
`nav_get_event_info()`	Get information about events from an EPG grid cache.
`nav_get_freq_info()`	Get information about a specific frequency.
`nav_get_num_of_events()`	Get the number of events meeting a specified criteria in an EPG grid cache.
`nav_get_num_chans()`	Retrieve the number of channels in a channel ring.
`nav_get_num_rings()`	Retrieve the number of channel rings.
`nav_get_preferences()`	Retrieve the Channel Manager's user preference settings.
`nav_get_ptr_chan_map()`	Retrieve a pointer to the Channel Manager's channel map.
`nav_get_ring_by_id()`	Retrieve a pointer to a channel ring's `ring` structure.
`nav_get_ring_by_name()`	Retrieve a pointer to a channel ring's `ring` structure.
`nav_get_signal_info()`	Retrieve information from the Tuner Driver for the currently tuned frequency.

Table 3-1 Summary of Navigation Management Functions (continued)

Function	Description
nav_get_network_time()	Get the system time as per the network tables
nav_getstat()	Pass an OEM defined getstat to a DBE hardware driver.
nav_init()	Initialize Navigation API.
nav_list_chans()	Retrieve a pointer to the list of channels in a channel ring.
nav_list_preceding_major_channels()	List the major channel numbers equal to or preceding the specified major channel number.
nav_list_preceding_minor_channels()	For a major channel number, list the minor channel numbers which equal or precede a specified minor channel number.
nav_list_rings()	Retrieve a pointer to the linked list of channel rings.
nav_list_succeeding_major_channels()	List the major channel numbers equal to or succeeding the specified major channel number.

Table 3-1 Summary of Navigation Management Functions (continued)

Function	Description
`nav_list_succeeding_minor_channels()`	For a major channel number, list the minor channel numbers which equal or succeed a specified minor channel number.
`nav_load_chan_map()`	Request the Channel Manager to load the channel map from the NRF file system.
`nav_lock_map()`	Lock the channel map from updates.
`nav_notify_asgn()`	Register for a notification on an asynchronous event.
`nav_notify_rmv()`	Remove a request for a notification on an asynchronous event.
`nav_remove_chan()`	Remove a channel from a favorite channel ring.
`nav_remove_digital_freq()`	Remove a digital frequency from the channel map.
`nav_remove_from_epg_cache()`	Remove the specified events from the event cache.
`nav_set_chan_map()`	Pass an initial channel map to the Channel Manager.
`nav_set_configuration()`	Set the Channel Manager's configuration settings.

Table 3-1 Summary of Navigation Management Functions (continued)

Function	Description
nav_set_cur_chan_next()	Change channels to the next channel in the current channel ring.
nav_set_cur_chan_num()	Change channels to a specified channel number.
nav_set_cur_chan_prec()	Change channels to the preceding channel in the current channel ring.
nav_set_cur_chan_prev_active()	Change channels to the channel map's previously active channel.
nav_set_cur_chan_ptr()	Change channels to a channel specified by a pointer to its `channel` structure.
nav_set_cur_chan_ring_prev_active()	Change channels to the current channel ring's previously active channel.
nav_set_cur_ring()	Change the current channel ring.
nav_set_preferences()	Set the Channel Manager's user preference settings.
nav_set_ring_name()	Set the name of a favorite channel ring.
nav_setstat()	Pass an OEM defined setstat to a DBE hardware driver.

Table 3-1 Summary of Navigation Management Functions (continued)

Function	Description
`nav_term()`	Terminate use of the Navigation API.
`nav_tune()`	Tune to a specified frequency.
`nav_update_minor_channel_list()`	Force Channel Manager to update the list of minor channels for a specified major channel number.
`nav_unlock_map()`	Unlock the channel map.

nav_abort()

Stop Decoding and Playout on the Current Channel

Syntax

```
#include <SPF/nav_api.h>

error_code nav_abort(void);
```

Description

If a digital channel is currently playing, this function stops decoding and playout of the channel's audio and video streams. If an analog channel is currently playing, this function has no effect.

Parameters

None.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

```
nav_set_cur_chan_next()
nav_set_cur_chan_num()
nav_set_cur_chan_prec()
nav_set_cur_chan_prev_active()
nav_set_cur_chan_ptr()
nav_set_cur_chan_ring_prev_active()
nav_set_cur_ring()
```

nav_add_chan()

Add a Channel to a Favorite Channel Ring

Syntax

```
#include <SPF/nav_api.h>

error_code nav_add_chan
(
   u_int32              ring_id,
   channel              *chan,
   channel              **new_chan
);
```

Description

Adds the channel specified by `chan` to the favorite channel ring specified by `ring_id`. A pointer to the `channel` structure for the new channel is returned in `*new_chan`.

Parameters

`ring_id`	The ring ID of an existing favorite channel ring. The ring ID is returned as an output parameter from the `nav_create_ring()` call.
`chan`	A pointer to a `channel` structure for the channel to be added. This parameter can point to a `channel` structure in the main channel ring or in any favorite channel ring.

WARNING

The validity of the `chan` pointer is not checked. The application must ensure a valid channel pointer is being used.

new_chan	A pointer to a channel pointer. A pointer to the channel structure for the newly added channel is returned in *new_chan.

API specific Errors

EOS_NAV_NOINIT	API not initialized.
EOS_ILLARG	Illegal parameter.
EOS_NAV_NERING	Non-existent ring.
EOS_NAV_AECHAN	Channel already exists in the specified ring.

Indirect Errors

Errors generated by _os_getstat() and _os_setstat().

See Also

nav_remove_chan()
nav_list_chans()
nav_get_num_chans()

nav_add_digital_freq()

Add a Digital Frequency to the Channel Map

Syntax

```
#include <SPF/nav_api.h>
#include <SPF/tuner.h>

error_code nav_add_digital_freq
(
   tuner_pb            *ptr_tpb,
   nav_notify          *notify,
   nav_status          *stat
);
```

Description

Adds a digital frequency to the list of frequencies that can be received. In response to this call, the Channel Manager will add the specified frequency to the channel map data structures, tune to this frequency, collect and parse the appropriate SI tables broadcast on this frequency, and update the channel map accordingly.

This call is mainly used in terrestrial broadcast environments as part of the frequency scan procedure that builds the channel map. In such environments the SI data on one frequency may not include channel map data for broadcasters located in the same geographic vicinity but assigned to other frequencies. For such environments each digital frequency that the television or set-top can receive must be explicitly added to the channel map via the `nav_add_digital_freq()` call. This procedure is described in further detail in Chapter 2.

In satellite or cable environments the SI data on each frequency typically includes information describing the channels for every frequency used in the system. In such cases the application tunes to an initial frequency using the `nav_tune()` call with the `SI_CHANNEL_LIST` option set. The Channel Manager will parse the SI data on the frequency in order to create the initial channel map. Thus, in satellite and cable environments the `nav_add_digital_freq()` call may never need to be issued by an application.

The nav_add_digital_freq() call can execute either synchronously or asynchronously. The call executes asynchronously when a non-null pointer to a nav_notify structure is passed for the ntfy parameter. When the ntfy parameter is NULL, the call executes synchronously, blocking until the SI data for the specified frequency has been acquired or an error has occurred. In this case the stat parameter is ignored.

When the ntfy parameter is not NULL, it must point to a nav_notify structure that specifies the type of asynchronous notification to be issued to the caller when the request has completed. In this case the stat parameter must point to a nav_status structure. The final status of the operation will be written into this structure before the requested notification is issued. As with all asynchronous requests to the Navigation API, if the return value of the call is SUCCESS, the requested notification will eventually be issued and the nav_status structure pointed to by the stat parameter will contain the final status of the request. However, if the call returns an immediate error code, the requested asynchronous notification will never be issued.

Parameters

ptr_tpb	A pointer to a tuner_pb structure containing the tuning parameters for the new frequency. This structure specifies the requested frequency and whether the frequency is analog or digital. The tuner_pb structure is defined in SPF/tuner.h.
ntfy	A pointer to a nav_notify structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the operation has completed. When the ntfy parameter is NULL, the call is executed synchronously. The nav_notify structure is defined in SPF/nav_api.h.
stat	A pointer to a nav_status structure. A status is written to *stat once the SI data for the new frequency has been added to

the channel map. If the call to `nav_add_digital_freq()` returns `SUCCESS`, the `nav_status` structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by `stat`. The `nav_status` structure is defined in `SPF/nav_api.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_create_analog_chan()`
`nav_remove_digital_freq()`

nav_add_to_epg_cache()

Add requested events to an EPG grid cache

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_add_to_epg_cache
(
   u_int32              cache_handle,
   u_int32              ring_id,
   u_int32              major_channel_num,
   u_int32              minor_channel_list[],
   int                  num_of_minor_channels,
   time_t               start_time,
   int                  duration_in_seconds,
   u_int32              flags,
   u_int32              *ptr_request_handle,
   nav_notify           *ntfy,
   nav_status           *stat
);
```

Description

Request the Channel Manager driver to retrieve events from the network and add them to an existing EPG cache. This call is typically asynchronous and returns a request handle which can be used to abort the call. However if the `ntfy` pointer is set to NULL, then the call is treated as a synchronous call.

The events to be retrieved are specified by the major channel number, minor channel number array, and time duration. The Channel Manager will tune to the frequency specified by `major_channel_num` to retrieve the events. When all the events specified are retrieved from the network and copied into the cache, or if the cache is full, then the call is terminated. The Channel Manager driver then notifies the application via either an event or a signal as specified in `ntfy`.

Parameters

`cache_handle` — Handle for an EPG cache returned by a previous `nav_create_epg_cache()` call

`ring_id` — The ID of the main channel ring containing the channels for which events are to be retrieved.

`major_channel_number`

Major channel number of the channels for which events are to be retrieved. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.

`minor_channel_list`

An array of minor channel numbers specifying the channels on which the events are to be retrieved. In DVB environments, the minor channel number is the same as the service id of the channel.

`num_of_minor_channels`

number of minor channels in the array above.

`start_time` — This field along with the `duration_in_seconds` field specifies the time range within which events are to be retrieved. All events for the specified channels, that are playing between `start_time` and `start_time+duration_in_seconds` are to be added to the cache.

`duration_in_seconds`

This field along with the `start_time` field specifies the time range within which events are to be retrieved. All events for the specified channels, that are playing between `start_time` and `start_time+duration_in_seconds` are to be added to the cache.

`flags`	The flags parmeter is a bitmask. Currently the only valid flag is `EPG_CACHE_LONG_TEXT` which indicates that the event description text associated with the events that are being to the cache, must also be stored. If the flag is not set, then the event description text is discarded.
`ptr_request_handle`	This field contains a pointer that will be used by the Channel Manager driver to return a handle to the `nav_add_to_epg_cache()` request. This handle can be used in the `nav_cancel_epg_retrieval_request()` to abort the `nav_add_to_epg_cache()` call.
`ntfy`	A pointer to a `nav_notify` structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the addition of the EPG data to the specified EPG cache has completed.
`stat`	A pointer to a `nav_status` structure. A status is written to *stat once the request for EPG data has been completed.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	One of the arguments passed was invalid.
`EOS_NPBNULL`	ntfy passed in was `NULL`.
`EOS_DEVBSY`	There is a previous request for EPG data for this cache which is still in progress.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_cancel_epg_retrieval_request()`

```
nav_remove_from_epg_cache()
nav_get_num_of_events()
nav_get_event_info()
nav_list_preceeding_minor_channels()
nav_list_succeeding_minor_channels()
nav_update_minor_channel_list()
```

nav_cancel_epg_retrieval_request()

Cancel an add to EPG cache request

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_cancel_epg_retrieval_request
(
    u_int32                 request_handle
);
```

Description

This function aborts a previous call to `nav_add_to_epg_cache()`.

Parameters

`request_handle`	Handle returrned by the `nav_add_to_epg_cache()` request.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	request_handle specified is `NULL` or invalid.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_add_to_epg_cache()`

nav_chan_set_userdef()

Set the User-defined Field of a chan_info Structure

Syntax

```
#include <SPF/nav_api.h>

error_code nav_chan_set_userdef
(
   channel               *chan,
   void                  *user_ptr
);
```

Description

Alters the contents of the user-defined field, `ch_user`, in the `chan_info` structure for the specified channel. This field may be used for application-defined purposes, and is initialized to `NULL` when a new channel is added to the channel map.

Parameters

`chan`	A pointer to a `channel` structure. The `ch_user` field of the `chan_info` structure pointed to by the `ch_info` field of the specified `channel` structure is updated.
`user_ptr`	The value to be stored in the `chan_info` structure's `ch_user` field.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_NECHAN`	Non-existent channel.
`EOS_NAV_NOTALLOWED`	Operation not allowed.

Indirect Errors

Errors generated by `_os_setstat()`.

nav_clear_chan_map()

Reset the Channel Map to an Empty State

Syntax

```
#include <SPF/nav_api.h>

error_code nav_clear_chan_map(void);
```

Description

Resets the channel map data structures to an empty state. For terrestrial broadcast environments a new channel map can then be created through a sequence of `nav_add_digital_freq()` and `nav_create_analog_chan()` calls to the Navigation API. For Satellite and Cable environments, a new channel map can be created by tuning to an initial frequency via the `nav_tune()` call with the `SI_CHANNEL_LIST` flag set.

One use of this function is to aid in creating a new channel map for when the set-top or television is moved from one geographic location to another. For example, when a family moves to a new city or state the previous channel map stored in the receiver's non-volatile memory may no longer be accurate. The Player Shell application can provide a menu option to perform a new frequency scan in order to create a valid channel map. In response to this menu selection, the Player Shell can delete the old channel map file from non-volatile memory by issuing the `nav_delete_chan_map_file()` call, reset the in-memory copy of the channel map to an empty state by issuing the `nav_clear_chan_map()` call, and then perform a frequency scan as described in Chapter 2 to build a new channel map.

Parameters

None.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

```
nav_add_digital_freq()
nav_create_analog_chan()
nav_delete_chan_map_file()
```

nav_create_analog_chan()

Create an Analog Channel in the Channel Map

Syntax

```
#include <SPF/nav_api.h>

error_code nav_create_analog_chan
(
    u_int32              freq_in_Hz,
    u_int32              major_ch_num,
    u_int32              minor_ch_num,
    channel              **new_chan
);
```

Description

Adds an analog channel to the main channel ring. A pointer to the `channel` structure for the new channel is returned in `*new_chan`.

This call is currently only supported in ATSC terrestrial broadcast environments. It is intended to be used as part of the frequency scan procedure that builds a channel map. In such environments the SI data broadcast on the digital frequencies assigned to broadcasters in a geographic area may not include channel map data for analog frequencies used by broadcasters located in the same geographic area. For example, a channel description for an analog channel broadcast by a station which has not yet begun to transmit on its new digital frequency may not appear in the SI data broadcast by any station in the vicinity. Therefore, the `nav_create_analog_chan()` call is provided to allow analog channels to be added to the channel map.

Note

For the ATSC terrestrial broadcast environment, the only way to add analog channels to the channel map is through the `nav_create_analog_chan()` call. That is, even if data describing an analog channel appears in a Virtual Channel Table (VCT) parsed by the ATSC Channel Manager, the channel won't be added to the channel map. This restriction is implemented in order to ensure that the signal for the analog channel in question is of high enough quality to produce an acceptable picture at the receiver. Thus, the application can control the signal quality of analog channels in the channel map through a combination of `nav_tune()`, `nav_get_signal_info()`, and `nav_create_analog_chan()` calls.

For those analog channels which have been added to the channel map with the `nav_create_analog_chan()` call and for which a channel description appears in a received VCT, the channel map will be updated to match the data for the channel from the received VCT.

Parameters

`freq_in_Hz`	The channel's frequency, specified in Hertz. For NTSC channels, this frequency should be 1.25 MHz above the lower edge of the 6 MHz band assigned to the channel. This frequency will be passed to the Tuner Driver in the `tuner_pb` structure during channel change operations to the newly added channel.
`major_ch_num`	The major channel number.

`minor_ch_num`	The minor channel number. For the ATSC Channel Manager, in the US implementation, the minor channel number for an analog channel must be 0. In DVB environments, the minor channel number is the same as the service id of the channel.
`new_chan`	A pointer to a `channel` pointer. A pointer to the `channel` structure for the newly added channel is returned in `*new_chan`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_add_digital_freq()`
`nav_destroy_analog_chan()`

nav_create_chan_map()

Create an empty channel map

Syntax

```
#include <SPF/nav_api.h>

error_code nav_create_chan_map(void);
```

Description

Creates an empty channel map. This call must be made before adding any frequencies to the Channel Manager. For terrestrial broadcast environments, this channel map can then be populated through a sequence of `nav_add_digital_freq()` and `nav_create_analog_chan()` calls to the Navigation API. For Satellite and Cable environments, this channel map can be populated by tuning to an initial frequency via the `nav_tune()` call with the `SI_CHANNEL_LIST` flag set.

One use of this function is to aid in creating a new channel map when the set-top or television is first installed in a home.

Parameters

None.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_add_digital_freq()`
`nav_create_analog_chan()`
`nav_delete_chan_map_file()`

nav_create_epg_cache

Allocate an EPG grid cache

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_create_epg_cache
(
    Epg_cache_params    ptr_cache_parameters,
    u_int32             *ptr_cache_handle
);
```

Description

Creates an empty EPG cache. Allocation of cache memory is done as specified by the cache parameters structure passed in. The call returns a cache handle which must be passed into subsequent calls using the cache.

Parameters

`ptr_cache_parameters`	
	A pointer to the cache parameters structure. This contains the type and size of the cache. This structure is defined in the header file epg_api.h
`ptr_cache_handle`	This is the returned value of the cache handle.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()` and `_os_srqmem()`.

See Also

nav_destroy_epg_cache()

nav_create_ring()

Create a New Favorite Channel Ring

Syntax

```
#include <SPF/nav_api.h>

error_code nav_create_ring
(
    u_char                  ring_type,
    char                    *ring_name,
    u_int32                 *ret_ring_id
);
```

Description

Creates a new channel ring with the ring name pointed to by `ring_name`. The `ring_type` parameter specifies the type of ring to be created. Currently, the only valid type which can be created with this call is `RING_TYPE_FAVORITE`. The ID of the new ring is returned in `*ret_ring_id`.

Parameters

`ring_type`	The type of ring to be created. This must be set to `RING_TYPE_FAVORITE`.
`ring_name`	A pointer to a character string specifying the name of the new channel ring. The string should be `NULL` terminated and the length of the string should be less than or equal to `RING_MAX_NAME` characters, including the `NULL` byte which terminates the string. The `RING_MAX_NAME` macro is defined in `SPF/nav_api.h`.
`ret_ring_id`	A pointer to a `u_int32` variable where the ID of the newly created ring is returned.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_AERING`	Duplicate ring name.

Indirect Errors

Errors generated by `_os_getstat()` and `_os_setstat()`.

See Also

`nav_add_chan()`
`nav_destroy_ring()`
`nav_set_ring_name()`
`nav_list_rings()`
`nav_remove_chan()`

nav_delete_chan_map_file()

Delete the Channel Map File From the NRF File System

Syntax

```
#include <SPF/nav_api.h>

error_code nav_delete_chan_map_file(void);
```

Description

Requests the Channel Manager to delete the channel map file from the NRF non-volatile RAM file system. The name of the channel map file is stored in the Channel Manager's device descriptor.

Parameters

None

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_clear_chan_map()`
`nav_load_chan_map()`

nav_destroy_analog_chan()

Remove an Analog Channel from the Channel Map

Syntax

```
#include <SPF/nav_api.h>

error_code nav_destroy_analog_chan
(
   u_int32              freq_in_Hz
);
```

Description

Deletes the analog channel on the specified frequency from the main channel ring and each favorite channel ring in which it appears. This call is currently only supported for ATSC terrestrial broadcast environments.

Parameters

`freq_in_Hz`	The channel's frequency, specified in Hertz. This frequency should be the same as that used in a previous `nav_create_analog_chan()` call to add the analog channel to the channel map.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_create_analog_chan()`
`nav_remove_digital_freq()`

nav_destroy_epg_cache

Free an EPG grid cache

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_destroy_epg_cache
(
  u_int32              cache_handle
);
```

Description

Deallocates all the memory associated with the specified EPG grid cache. Any asynchronous EPG calls such as `nav_add_to_epg_cache()` using this cache handle are terminated.

Parameters

`cache_handle`	EPG cache handle returned by `nav_create_epg_cache()`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Invalid cache handle specified.

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`nav_create_epg_cache()`

`nav_flush_epg_cache()`

Syntax

```
#include <SPF/nav_api.h>

error_code nav_destroy_ring
(
   u_int32                ring_id
);
```

Description

Destroys the favorite channel ring specified by `ring_id`.

Parameters

`ring_id`	The ring ID of an existing favorite channel ring. The ring ID is returned as an output parameter from the `nav_create_ring()` call.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_NAV_NERING`	Non-existent ring.
`EOS_NAV_INUSE`	The specified ring is the current channel ring.

Indirect Errors

Errors generated by `_os_getstat()` and `_os_setstat()`.

See Also

`nav_clear_chan_map()`
`nav_list_rings()`

nav_flush_epg_cache()

Flush all events from the specified EPG grid cache

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_flush_epg_cache
(
    u_int32             cache_handle
);
```

Description

This function removes all the events from a specified cache. After this call, the cache is in an empty state.

Parameters

`cache_handle`	Identifier for the specified cache.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized
`EOS_ILLARG`	Invalid cache handle specified

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_create_epg_cache()`

`nav_destroy_epg_cache()`

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_chan
(
    u_int32             ring_id,
    u_int32             major_ch_num,
    u_int32             minor_ch_num,
    channel             **ret_chan_ptr
);
```

Description

Returns a pointer to the specified channel’s `channel` structure in the specified channel ring. The channel is specified by its channel number, and the channel ring is specified by its ring ID.

Parameters

`ring_id`	The ring ID of the ring containing the channel whose `channel` pointer is to be returned. For favorite channel rings the ring ID is returned as an output parameter from the `nav_create_ring()` call. The ring ID of the main channel ring is zero.
`major_ch_num`	The major channel number of the requested channel. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.
`minor_ch_num`	The minor channel number of the requested channel.

`ret_chan_ptr`	A pointer to a `channel` pointer. A pointer to the specified channel's `channel` structure is returned in `*ret_chan_ptr`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_NERING`	Non-existent ring.
`EOS_NAV_NECHAN`	Non-existent channel.

Indirect Errors

Errors generated by `_os_getstat()`.

See Also

`nav_set_cur_chan_ptr()`
`nav_remove_digital_freq()`
`nav_get_configuration()`
`nav_get_num_chans()`
`nav_get_preferences()`
`nav_get_ptr_chan_map()`

nav_get_configuration()

Retrieve the Channel Manager's Configuration Settings

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_configuration
(
  nav_config           *config
);
```

Description

Retrieves the current values of the Channel Manager's configuration settings.

Parameters

`config`	A pointer to a `nav_config` structure. The current configuration settings are returned in `*config`. The `nav_config` structure is defined in `SPF/nav_api.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_set_configuration()`

nav_get_cur_chan()

Retrieve a Pointer to the Current Channel's channel Structure

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_cur_chan
(
    channel              **ret_chan_ptr
);
```

Description

Returns a pointer to the `channel` structure for the current channel in the current channel ring.

Parameters

`ret_chan_ptr`	A pointer to a `channel` pointer. A pointer to the current channel's `channel` structure is returned in `*ret_chan_ptr`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_NOCURRENT`	No current channel.

Indirect Errors

Errors generated by `_os_getstat()`.

See Also

`nav_set_cur_chan_ptr()`
`nav_remove_digital_freq()`
`nav_get_num_chans()`
`nav_get_preferences()`
`nav_get_ptr_chan_map()`
`nav_get_chan()`

nav_get_cur_chan_num()

Get the major/minor channel number for the current channel

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_cur_chan_num
(
  u_int32             *major_chan_num_ptr,
  u_int32             *minor_chan_num_ptr
);
```

Description

Get the major and minor channel numbers for the current channel.

Parameters

`major_chan_num_ptr`

A pointer to the location where the major channel number will be returned. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.

`minor_chan_num_ptr`

A pointer to the location where the minor channel number will be returned. In DVB environments, the minor channel number is the same as the service id of the channel.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_NOCURRENT`	No current channel.

Indirect Errors

Errors generated by `_os_getstat()`.

See Also

`nav_set_cur_chan_ptr()`
`nav_remove_digital_freq()`
`nav_get_num_chans()`
`nav_get_preferences()`
`nav_get_ptr_chan_map()`
`nav_get_chan()`

nav_get_cur_ring()

Retrieve a Pointer to the Current Channel Ring's ring Structure

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_cur_ring
(
   ring                    **ret_ring_ptr
);
```

Description

Returns a pointer to the current channel ring.

Parameters

`ret_ring_ptr`	A pointer to a `ring` pointer. A pointer to the current channel ring's `ring` structure is returned in `*ret_ring_ptr`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_NOCURRENT`	No current channel ring.

Indirect Errors

Errors generated by `_os_getstat()`.

See Also

`nav_get_num_chans()`
`nav_get_preferences()`

nav_get_current_event()

Get a pointer to the current event on the current channel

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_get_current_event
(
   Epg_event             pp_curr_event
);
```

Description

This call returns a pointer to the event currently playing on the channel currently being viewed.

Note

The driver does not return a pointer to its current event structure. Instead, it allocates separate memory for the current event structure on a per-path basis. This is because the current event may be updated at any time. Therefore driver maintains a separate copy of the current event structure which is updated as per the network SI tables. Then when the application issues the `nav_get_current_event()` call, the driver updates the current event memory allocated for that path. The application can track changes to the current event by using the nav_notify_asgn() call and requesting the notification `NAV_NTFY_ON_CURR_EVENT_AVAILABLE`. When it receives the notification it can reissue the na_get_current_event() call to update its copy of the current event structure.

Parameters

`pp_curr_event`	pointer to the returned pointer to the current event structure.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Invalid event pointer specified.

Indirect Errors

Errors generated by `_os_getstat()`.

See Also

`nav_notify_asgn()`

nav_get_event_info()

Get information about events on a channel from an EPG cache

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_get_event_info
(
   u_int32              cache_handle,
   u_int32              ring_id,
   u_int32              major_channel_num,
   u_int32              minor_channel_num,
   time_t               start_time,
   u_int32              duration_in_seconds,
   Epg_event            event_ptr_list[],
   int                  *ptr_num_of_events
);
```

Description

This call is used to get specified events from an EPG cache which has been previously filled using the `nav_add_to_epg_cache()` call. This call allows the application to specify selection criteria based on which pointers to events from the cache are returned in the `event_ptr_list` array. Note that this call does not retrieve events from the network. It simply copies pointers to selected events (based on time and channel) that are already in the cache into the `event_ptr_list` array.

Parameters

`cache_handle`	handle to the EPG cache returned by the `nav_create_epg_cache()` call.
`ring_id`	Id of the main ring containing the channels for which the events are to be returned.

`major_ch_num`	The major channel number of the channel for which the events are to be returned. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.
`minor_ch_num`	The minor channel number of the channel for which the events are to be returned. For the ATSC Channel Manager, in the US implementation, the minor channel number for an analog channel must be 0. In DVB environments, the minor channel number is the same as the service id of the channel.
`start_time`	Beginning time of the specified time span within which events are to be returned.
`duration_in_seconds`	Number of seconds in the specified time span within which events are to be returned.
`event_ptr_list`	Pointer to array of event pointers. This must be allocated by the application. The application can predetermine the size of this array via the `nav_get_num_of_events()`.
`ptr_num_of_events`	Pointer to the number of events. The application must set this to the size of the array of event pointers passed in. The driver will set this to the actual number of event pointers returned.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Invalid argument passed to function.

Indirect Errors

Errors generated by `_os_getstat()`

See Also

`nav_get_num_of_events()`

`nav_get_current_event()`

nav_get_freq_info()

Get information about a frequency

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_freq_info
(
   ring                      *ring_ptr,
   u_int32                   freq,
   u_int32                   *ptr_transport_id,
   u_int32                   *ptr_major_ch_num,
   u_int32                   *ptr_is_digital
);
```

Description

Returns information about a specified frequency. This may be used in the initial frequency scanning procedure for a number of reasons for e.g. in the case of the ATSC Channel Manager, to detect duplicate transport streams on the network.

Parameters

`ring_ptr`	Pointer to the ring on which the frequency is located. This must point to a Main Channel Ring.
`freq`	Frequency for which information is requested. For the DVB environment this value is specified in kHz. For the ATSC environment, this value is specified in Hz.
`ptr_transport_id`	Pointer to the location where the Channel Manager will store the transport id for the frequency.
`ptr_major_ch_num`	Pointer to the location where the Channel Manager will store the major channel number for the frequency.

`ptr_is_digital`	If the frequency is digital, the Channel Manager will set the location pointed by this field to 1 else it will be set to 0.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_getstat()`

See Also

`nav_add_digital_freq()`
`nav_create_analog_chan()`

nav_get_num_of_events()

Get information about events on a channel from an EPG grid cache

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_get_num_of_events
(
    u_int32             cache_handle,
    u_int32             ring_id,
    u_int32             major_channel_num,
    u_int32             minor_channel_num,
    time_t              start_time,
    u_int32             duration_in_seconds,
    int                 *ptr_num_of_events
);
```

Description

Returns the number of events in an EPG cache that meet a specific criteria based on channel and time. This call is used before the `nav_get_event_info()` call to determine the size of the array of event pointers that needs to be allocated and passed into that call.

Parameters

`cache_handle`	handle to the EPG cache returned by the `nav_create_epg_cache()` call.
`ring_id`	Id of the main ring containing the channels for which the events are to be returned.
`major_ch_num`	The major channel number of the channel for which the events are to be selected. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains

	the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.
minor_ch_num	The minor channel number of the channel for which the events are to be selected. For the ATSC Channel Manager, in the US implementation, the minor channel number for an analog channel must be 0. In DVB environments, the minor channel number is the same as the service id of the channel.
start_time	Beginning time of the time span within which events are to be selected.
duration_in_seconds	Number of seconds in the time span within which events are to be selected.
ptr_num_of_events	Pointer to the number of events. The driver will set this to the actual number of events which meet the criteria.

API specific Errors

EOS_NAV_NOINIT	API not initialized
EOS_ILLARG	Invalid argument passed to function

Indirect Errors

Errors generated by `_os_getstat()`

See Also

`nav_get_event_info()`

nav_get_num_chans()

Retrieve the Number of Channels in a Channel Ring

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_num_chans
(
   u_int32              ring_id,
   u_int16              flags,
   u_int32              *count
);
```

Description

Returns the number of channels in the specified channel ring that match the criteria specified by the `flags` parameter.

The `flags` parameter is a bitmask which specifies the type of channels to be counted by this call. If `flags` is zero, all channels are counted. Valid flags are `NAV_FLAG_HIDDEN` and `NAV_FLAG_NOT_HIDDEN`. If the value of `flags` is `NAV_FLAG_HIDDEN`, then only channels that have the `CHAN_HIDDEN` bit set in the `chan_flags` field of the channel's `chan_info` structure are counted. Similarly, if the value of `flags` is `NAV_FLAG_NOTHIDDEN`, then only channels that have the `CHAN_HIDDEN` bit not set in the `chan_flags` field of the channel's `chan_info` structure are counted.

Parameters

`ring_id`	The ring ID of the ring whose count of channels is to be retrieved. For favorite channel rings the ring ID is returned as an output parameter from the `nav_create_ring()` call. The ring ID of the main channel ring is zero.
`flags`	A `u_int16` bitmask that specifies the type of channels to count.

count	A pointer to a u_int32 variable. The number of channels is returned in *count.

API specific Errors

EOS_NAV_NOINIT	API not initialized.
EOS_ILLARG	Illegal parameter.
EOS_NAV_NERING	Non-existent ring.

Indirect Errors

Errors generated by _os_getstat()

See Also

nav_get_num_rings()

nav_get_num_rings()

Retrieve the Number of Channel Rings

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_num_rings
(
   u_int32              *count
);
```

Description

Returns the number of channel rings maintained by the Channel Manager Driver. This will always be the current number of favorite channel rings plus one (for the main channel ring).

Parameters

`count`	A pointer to a `u_int32` variable. The number of channel rings is returned in `*count`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized
`EOS_ILLARG`	Illegal parameter

Indirect Errors

Errors generated by `_os_getstat()`

See Also

`nav_list_rings()`

nav_get_preferences()

Retrieve the Channel Manager's User Preference Settings

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_preferences
(
   ch_user_pref         *prefs
);
```

Description

Retrieves the current user preference settings (such as language preferences for audio track selection) from the Channel Manager Driver.

Parameters

`prefs`	A pointer to a `ch_user_pref` structure. The current user preferences settings are returned in `*ch_user_pref`. The `ch_user_pref` structure is defined in `SPF/nav_api.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized
`EOS_ILLARG`	Illegal parameter

Indirect Errors

Errors generated by `_os_getstat()`

See Also

`nav_set_preferences()`

nav_get_ptr_chan_map()

Retrieve a Pointer to the Channel Manager's Channel Map

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_ptr_chan_map
(
   void                  **chan_map,
   u_int32               *size
);
```

Description

Retrieves a pointer to the channel map data area allocated by the Channel Manager Driver. An application can use this function to obtain direct access to the channel map data structures or to retrieve a pointer to the channel map in order to copy the channel map to non-volatile memory before the set-top or television is powered down.

> **Note**
>
> If the Channel Manager's source code has been compiled with the `NRF` preprocessor macro defined and the first process to initialize the Channel Manager has passed a value of `TRUE` for the `*use_NRF` parameter to `nav_init()`, then the Channel Manager will automatically save the channel map when the last process using the Channel Manager issues the `nav_term()` call. In such cases, the application does not need to copy the channel map to non-volatile memory.

The permissions on the channel map's memory are set so that applications which use the Navigation API can read the memory but are prevented from writing to it.

Parameters

chan_map	A pointer to a void pointer. A pointer to the channel map is returned in *chan_map.
size	A pointer to a u_int32. The total size (in bytes) of the channel map data is returned in *size.

API specific Errors

EOS_NAV_NOINIT	API not initialized
EOS_ILLARG	Illegal parameter

Indirect Errors

Errors generated by _os_setstat()

See Also

nav_lock_map()
nav_set_chan_map()
nav_unlock_map()

nav_get_ring_by_id()

Retrieve a Pointer to a Channel Ring's ring Structure

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_ring_by_id
(
    u_int32             ring_id,
    ring                **ret_ring_ptr
);
```

Description

Retrieves a pointer to the `ring` structure for the specified channel ring. The channel ring is specified by its ring ID.

Parameters

`ring_id`	The ring ID of the ring for which the address of the `ring` structure is to be retrieved. For favorite channel rings the ring ID is returned as an output parameter from the `nav_create_ring()` call. The ring ID of the main channel ring is zero.
`ret_ring_ptr`	A pointer to a `ring` pointer. A pointer to the channel ring's `ring` structure is returned in `*ret_ring_ptr`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized
`EOS_ILLARG`	Illegal parameter
`EOS_NAV_NERING`	Non-existent ring

Indirect Errors

Errors generated by `_os_getstat()`

See Also

nav_get_ring_by_name()
nav_list_rings()

nav_get_ring_by_name()

Retrieve a Pointer to a Channel Ring's ring Structure

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_ring_by_name
(
   char                    *ring_name,
   ring                    **ret_ring_ptr
);
```

Description

Retrieves a pointer to the `ring` structure for the specified channel ring. The channel ring is specified by its ring name.

Parameters

`ring_name`	A pointer to the character string specifying the name of the requested channel ring. The string should be `NULL` terminated.
`ret_ring_ptr`	A pointer to a `ring` pointer. A pointer to the channel ring's `ring` structure is returned in `*ring_ptr`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized
`EOS_ILLARG`	Illegal parameter
`EOS_NAV_NERING`	Non-existent ring

Indirect Errors

Errors generated by `_os_getstat()`

See Also

`nav_get_ptr_chan_map()`

nav_list_rings()

nav_get_signal_info()

Retrieve Information From the Tuner Driver for the Currently Tuned Frequency

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_signal_info
(
  void                *signal_info
);
```

Description

Retrieves information on the currently tuned frequency from the Tuner Driver.

The type of information retrieved by this function is dependent on the Tuner Driver. Typical information may include the signal strength, whether the frequency is analog or digital, and bit-error rates for digital frequencies.

In digital terrestrial broadcast receivers, the set of receivable frequencies will depend on the geographic location (e.g. city and state) of the receiver. For such environments the `nav_get_signal_info()` call can be used as part of the procedure to determine the receivable frequencies and create the initial channel map. In such environments, an application can tune to each possible frequency by issuing the `nav_tune()` call, measure the signal strength on the frequency by issuing the `nav_get_signal_info()` call, and, if the signal information indicates that a broadcaster assigned to that frequency is within range, add the frequency to the list of receivable frequencies by issuing the `nav_add_digital_freq()` or `nav_create_analog_chan()` call.

In direct broadcast satellite receivers, the `nav_get_signal_info()` call can be used to obtain information on the signal strength in order to help aim the satellite dish during installation procedures. For example, an application can obtain the signal strength and display it on the screen as a bar-graph value in real time as the satellite dish is being aimed.

Parameters

`signal_info`	A `void` pointer that points to the memory location where the current signal information should be returned. The application and Tuner Driver must agree on the format of the information returned in this memory. Typically the OEM will define a structure for this purpose.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized
`EOS_ILLARG`	Illegal parameter

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`nav_add_digital_freq()`
`nav_create_analog_chan()`
`nav_tune()`

nav_get_network_time()

Get the system time as per the network

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_get_system_time
(
    time_t              *ptr_utc_time,
);
```

Description

This function gets the current system time from the network SI tables in Universal Time Coordinated (UTC or GMT) time. The application should not rely on the network time being constantly updated and available. This is because the network time will not be available when the user is tuned to an analog channel. Typically the application should request this value and then set the local clock on the Set-top Box using the time retrieved. This can be done via the _os_setime() call.

If the network time table is not available, or if the user is tuned to an analog channel, this call will return an `EOS_CH_NO_NTWK_TIME` error.

Parameters

`ptr_utc_time`	Pointer to structure which will receive the Universal Time Coordinated (GMT) time.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Invalid value passed into function.
`EOS_CH_NO_NTWK_TIME`	Time information not available currently.

Indirect Errors

Errors generated by `_os_getstat()`.

nav_getstat()

Pass an OEM Defined Getstat to a DBE Hardware Driver

Syntax

```
#include <SPF/nav_api.h>

error_code nav_getstat
(
  void                *pb
);
```

Description

Passes an OEM defined getstat request to the Tuner Driver or Conditional Access Driver. The Channel Manager will route the getstat request to the proper DBE driver.

The `pb` parameter must point to a memory area which contains a valid `spf_ss_pb` structure at the start of the memory. The Channel Manager Driver will use the protocol ID value in the upper 16 bits of the `code` field of this `spf_ss_pb` structure to route the getstat request to the proper DBE driver. The recommended method for using this function is to define a structure whose first field is an `spf_ss_pb` structure and pass the address to this enclosing structure as the `pb` parameter. The enclosing structure can contain the additional fields necessary to support the OEM defined getstat request.

The bottom 16 bits of the `code` field of the `spf_ss_pb` structure define the specific getstat request being issued. In order to prevent conflicts with getstat codes defined by the DBE package, these bits should contain a value of hex 100 or greater.

Note

The Channel Manager Driver contains a variable that records the currently tuned frequency. In order to maintain the consistency of this variable, OEM defined getstat requests implemented by the Tuner Driver should not result in a frequency change. To accomplish an explicit frequency change, an application should instead issue the `nav_tune()` call.

Parameters

pb	A pointer to the getstat parameter block. The getstat parameter block memory should contain an `spf_ss_pb` structure as its first field. The `spf_ss_pb` structure is defined in `SPF/spf.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_getstat()`.

See Also

`nav_setstat()`
`nav_tune()`

nav_init()

Initialize the Navigation API

Syntax

```
#include <SPF/nav_api.h>

error_code nav_init
(
   BOOLEAN              *use_NRF
);
```

Description

Initializes the Navigation API and opens a path to the Channel Manager Driver. Every process which uses the Navigation API must perform a `nav_init()` call as its first call to the Navigation API.

Parameters

`use_NRF`	A pointer to a `boolean` data type which specifies whether or not the Channel Manager should use the NRF non-volatile RAM file system for saving and restoring the channel map data structures and user configuration information. The `boolean` data type pointed to by `use_NRF` is both an input and an output parameter to `nav_init()`. If no process currently has a path open to the Channel Manager Driver, then the first process to successfully perform a `nav_init()` call can request the Channel Manager to use NRF by passing a value of `TRUE` for `*use_NRF`, or to not use NRF by passing a value of `FALSE` for `*use_NRF`. The value returned in `*use_NRF` indicates whether or not the Channel Manager has been configured to use NRF. A value of

TRUE is returned if the Channel Manager will use NRF, otherwise a value of FALSE is returned.

For a successful `nav_init()` call, the returned value of `*use_NRF` will match the requested value except under the following two cases:

A path to the Channel Manager is already open when a process performs a `nav_init()` call. In this case the input value of `*use_NRF` is ignored and the returned value indicates whether the Channel Manager is configured (via a previously-opened path) to use NRF.

A process specifies a value of TRUE for `*use_NRF`, but the Channel Manager Driver was not compiled to support NRF. In this case the returned value for `*use_NRF` will be FALSE.

.API specific Errors

`EOS_NAV_ISINIT`	API already initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_open()` and `_os_setstat()`.

See Also

`nav_term()`

nav_list_chans()

Retrieve a Pointer to the List of Channels in a Channel Ring

Syntax

```
#include <SPF/nav_api.h>

error_code nav_list_chans
(
  u_int32             ring_id,
  channel             **channel_list,
  u_int32             *count
);
```

Description

Retrieves a pointer to the linked list of `channel` structures for the specified channel ring. The `channel` structures are arranged in a `NULL` terminated doubly linked list that is sorted in increasing order by channel number. The total number of channels in the specified channel ring is returned in `*count`.

Note

In ATSC environments, channels may dynamically disappear and reappear in the SI data for a broadcaster as the broadcaster switches between several SDTV programs and a single HDTV program. A channel which disappears is marked as deleted in the channel map, but is not initially removed from favorite channel rings. This allows the channel to remain in favorite channel rings in case it reappears at a later time.

When the Channel Manager needs a data structure for a new channel, but has already allocated the maximum number of allowed channels as specified by its configuration settings, it will reclaim a channel marked as deleted. In this case, the channel which has been deleted the longest is reclaimed.

When traversing a channel list, applications can skip deleted channels by examining the `chan_flags` field of the `chan_info` structure pointed to by the `channel` structure's `ch_info` field. Deleted channels are indicated by the `CHAN_DELETED` bit being set to one in the `chan_flags` field.

Parameters

`ring_id`	The ring ID of the ring for which the pointer to the `channel` list is to be retrieved. For favorite channel rings the ring ID is returned as an output parameter from the `nav_create_ring()` call. The ring ID of the main channel ring is zero.
`channel_list`	A pointer to a `channel` pointer. A pointer to the beginning of the `channel` list for the specified channel ring is returned in `*channel_list`.
`count`	A pointer to a `u_int32` variable. The total number of channels in the specified ring is returned in `*count`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_NERING`	Non-existent ring.

Indirect Errors

Errors generated by `_os_getstat()`.

See Also

`nav_list_rings()`

nav_list_preceding_major_channels()

List major channel numbers preceding the specified channel

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_list_preceding_major_channels
(
  u_int32             ring_id,
  u_int32             major_channel_num,
  u_int32             major_channel_list[],
  int                 *ptr_major_channel_count
);
```

Description

This function lists all the major channel numbers that equal to, or precede the specified major channel number.

Parameters

`ring_id` Id of the main ring ring containing the specified channels.

`major_channel_num`

Specified major channel number. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.

`major_channel_list`

Pointer to array of returned major channel numbers. Memory for the array must be allocated by the application.

`ptr_major_channel_count`

Pointer to number of elements in the array. The application sets this field to the number of elements in the array passed in. The driver will set this field to the actual number of entries returned.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	An invalid argument was passed to the function.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_list_preceding_minor_channels()`

`nav_list_succeeding_major_channels()`

`nav_list_succeeding_minor_channels()`

`nav_update_minor_channel_list()`

nav_list_preceding_minor_channels()

List minor channel numbers for a particular major channel preceding a specified minor channel

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_list_preceding_minor_channels
(
   u_int32             ring_id,
   u_int32             major_channel_num,
   u_int32             minor_channel_num,
   u_int32             minor_channel_list[],
   int                 *ptr_minor_channel_count
);
```

Description

For a specified major channel, this funtion will return a list of minor channels which are equal to, or precede the specified minor channel.

Parameters

`ring_id`
: Id of the main ring ring containing the specified channels.

`major_channel_num`
: Specified major channel number. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.

`minor_chan_num`
: Specified minor channel number.

`minor_channel_list`

Pointer to array of returned minor channel numbers. Memory for the array must be allocated by the application. In DVB environments, the minor channel number is the same as the service id of the channel.

`ptr_minor_channel_count`

Pointer to number of elements in the array. The application sets this field to the number of elements in the array passed in. The driver will set this field to the actual number of entries returned.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	An invalid argument was passed to the function.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_list_preceding_major_channels()`

`nav_list_succeeding_major_channels()`

`nav_list_succeeding_minor_channels()`

`nav_update_minor_channel_list()`

nav_list_rings()

Retrieve a Pointer to the Linked List of Channel Rings

Syntax

```
#include <SPF/nav_api.h>

error_code nav_list_rings
(
   ring                   **ret_ring_list,
   u_int32                *count
);
```

Description

Retrieves a pointer to the list of channel rings. The `ring` structures are arranged in a `NULL` terminated doubly linked list. The `ring` structure for the first channel ring in the list is always the main channel ring. The total number of rings is returned in `*count`.

Parameters

`ret_ring_list`	A pointer to a `ring` pointer. A pointer to the beginning of the channel ring list is returned in `*ret_ring_list`.
`count`	A pointer to a `u_int32` variable. The total number of channel rings is returned in `*count`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `ite_chn_map_get()`.

See Also

`nav_list_rings()`

nav_list_succeeding_major_channels()

List major channel numbers succeeding the specified channel

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_list_succeeding_major_channels
(
   u_int32             ring_id,
   u_int32             major_channel_num,
   u_int32             major_channel_list[],
   int                 *ptr_major_channel_count
);
```

Description

This function lists all the major channel numbers that equal to, or succeed the specified major channel number.

Parameters

`ring_id` Id of the main ring ring containing the specified channels.

`major_channel_num`

Specified major channel number. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.

`major_channel_list`

Pointer to array of returned major channel numbers. Memory for the array must be allocated by the application.

`ptr_major_channel_count`

Pointer to number of element in the array. The application sets this field to the number of elements in the array passed in. The driver will set this field to the actual number of entries returned.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	An invalid argument was passed to the function.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_list_preceding_major_channels()`

`nav_list_preceding_minor_channels()`

`nav_list_succeeding_minor_channels()`

`nav_update_minor_channel_list()`

nav_list_succeeding_minor_channels()

List minor channel numbers for a particular major channel succeeding a specified minor channel

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_list_succeeding_minor_channels
(
   u_int32              ring_id,
   u_int32              major_channel_num,
   u_int32              minor_channel_num
   u_int32              minor_channel_list[],
   int                  *ptr_minor_channel_count
);
```

Description

For a specified major channel, this funtion will return a list of minor channels which are equal to, or succeed the specified minor channel.

Parameters

`ring_id`
: Id of the main ring ring containing the specified channels.

`major_channel_num`
: Specified major channel number. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.

`minor_chan_num`
: Specified minor channel number.

`minor_channel_list`

Pointer to array of returned minor channel numbers. Memory for the array must be allocated by the application. In DVB environments, the minor channel number is the same as the service id of the channel.

`ptr_minor_channel_count`

Pointer to number of elements in the array. The application sets this field to the number of elements in the array passed in. The driver will set this field to the actual number of entries returned.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	An invalid argument was passed to the function.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_list_preceding_major_channels()`

`nav_list_preceding_minor_channels()`

`nav_list_succeeding_major_channels()`

`nav_update_minor_channel_list()`

nav_load_chan_map()

Request the Channel Manager to Load the Channel Map From the NRF File System

Syntax

```
#include <SPF/nav_api.h>

error_code nav_load_chan_map(void);
```

Description

Requests the Channel Manager to load the channel map from the NRF non-volatile RAM file system. The name of the channel map file is stored in the Channel Manager's device descriptor.

The Channel Manager Driver can be conditionally compiled to include support for automatically saving and retrieving the channel map to a file in the NRF non-volatile RAM file system. If this support has been compiled into the driver and if the Channel Manager was initialized (via the `nav_init()` call) to use NRF, then the Channel Manager will automatically save the channel map file to the NRF file system when the last process using the Navigation API issues the `nav_term()` call. When the set-top or television is powered back up, an application can request the Channel Manager to load this channel map file by issuing the `nav_load_chan_map()` call. In order for the Channel Manager to support this automatic saving and restoring of the channel map, the NRF file manager must be included in the system and the NRF preprocessor macro must be defined when compiling the Channel Manager.

An alternate method for saving and restoring the channel map is provided by the `nav_get_ptr_chan_map()` and `nav_set_chan_map()` calls. These calls can be used in systems which do not include the NRF file manager.

The `nav_load_chan_map()` call can only be successfully issued after the Channel Manager has been initialized (via the `nav_init()` call), but before the Channel Manager has created a channel map. The Channel Manager will create a channel map in response to a successful call to any of the following Navigation API functions: `nav_add_digital_freq()`, `nav_create_analog_chan()`, `nav_load_chan_map()`, or `nav_set_chan_map()`.

Parameters

None

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_add_digital_freq()`
`nav_create_analog_channel()`
`nav_delete_chan_map_file()`
`nav_set_chan_map()`

nav_lock_map()

Lock the Channel Map from Updates

Syntax

```
#include <SPF/nav_api.h>

error_code nav_lock_map(void);
```

Description

Locks the channel map from being updated. Normally, the channel map is updated by the Channel Manager driver as new SI data is received, or in response to certain calls to the Navigation API. (For example, many calls to the Navigation API result in one or more of the fields in a channel map data structure being changed.) A process can lock the channel map from updates due to new SI data being received or Navigation API calls being issued by other processes by calling the `nav_lock_map()` function. Locking the channel map allows it to remain in a consistent state as its data structures are being examined and traversed, or as the channel map is being copied by an application to NVRAM. After locking and examining the channel map, a process should issue the `nav_unlock_map()` call in order to allow further updates to the channel map.

When necessary, the Navigation API will lock the channel map on entry to one of its functions and unlock the map before returning to the calling application. Thus, it is only necessary for applications to call `nav_lock_map()` if the channel map must remain unchanged across multiple calls to the Navigation API or if the application wishes to directly examine or copy the channel map data structures.

An application may issue the `nav_lock_map()` call and then call a Navigation API function which also locks the channel map on the application's behalf. Therefore, once the channel map is locked, the Channel Manager maintains a count of how many times a lock request was issued on behalf of the process with the current lock. The channel map is only unlocked when the corresponding number of unlock requests are issued. Therefore, an application should issue an explicit `nav_unlock()` call for each explicit `nav_lock()` call that it has issued.

WARNING
In response to the `nav_add_digital_freq()` call or any Navigation API call which requests a channel change operation, the Channel Manager may need to tune to the appropriate frequency and collect updated SI data. Thus, if any such call is issued while the channel map is locked, the channel map may actually change while it is locked.

Parameters

None

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat() and _os_getstat()`

See Also

`nav_unlock_map()`

nav_notify_asgn()

Register for a Notification on an Asynchronous Event

Syntax

```
#include <SPF/nav_api.h>

error_code nav_notify_asgn
(
    nav_notify_event    event,
    nav_notify          *ntfy,
    nav_status          *stat
);
```

Description

Registers the calling application to be notified upon a specific asynchronous Channel Manager event. The `event` parameter specifies the specific event for which a notification is desired.

The following events are currently supported:

`NAV_NTFY_ON_MAP_MODIFIED`

`NAV_NTFY_ON_CURR_EVENT_AVAILABLE`

The `NAV_NTFY_ON_MAP_MODIFIED` event occurs when all sections of a new version of an SI table parsed by the Channel Manager have been received and parsed, and the channel map has been updated accordingly.

In ATSC terrestrial broadcast environments, the set of active channels may change fairly often as broadcasters dynamically reallocate bandwidth between multiple SDTV channels and a single HDTV channel. The `nav_notify_asgn()` call can be used following a channel change request to a channel that is not currently active in order for an application to receive a notification when the specified channel does become active. In response to a channel change request to an inactive channel, the Channel Manager will map the desired channel number to a frequency (if possible), tune to this frequency, update its channel map according to the current SI data for the frequency, and search for the specified channel number in the updated SI data. If no channel with the specified channel number exists, the Channel Manager will return an `EOS_CHAN_NOT_ACTIVE` error code as the status of the channel change request. At this point, the application can blank the screen, mute the audio, display the selected channel number in a corner of the screen, and register for a notification to be issued when the channel map has been modified. When the notification is received, the application can re-attempt the channel change, and, if successful, remove the channel number from the corner of the screen, un-blank the screen, and un-mute the audio. Thus, from the viewer's perspective, the specified channel will automatically start playing once the channel does become active.

The `NAV_NTFY_ON_CURR_EVENT_AVAILABLE` notification is sent when new information about the currently playing event on the currently viewed channel becomes available. When this call is made, if information about the current event is already present then the notification is sent immediately. However note that this notification is a persistent notification, i.e the request remains active until explicitly cancelled by the application. Hence the application needs to issue this call only once and it will keep receiving notifications when the current event structure changes. On receiving this notification, the application may get a pointer to the current event via the `nav_get_current_event()` call.

Parameters

`event`	A `nav_notify_event` enumerated type specifying the asynchronous event for which the application wishes to be notified. Currently the only valid values for this parameter are `NAV_NTFY_ON_MAP_MODIFIED` and `NAV_NTFY_ON_CURR_EVENT_AVAILABLE`. The `nav_notify_event` enumerated type is defined in `SPF/nav_api.h`.
`ntfy`	A pointer to a `nav_notify` structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the specified event has occurred. The `nav_notify` structure is defined in `SPF/nav_api.h`.

stat	A pointer to a nav_status structure. A status associated with the event is written in *stat once the Channel Manager event has occurred. Currently for the events defined, the status returned in *stat will always be SUCCESS. If the call to nav_notify_asgn() returns SUCCESS, the nav_status structure must remain allocated by the application until either the requested notification is received or the nav_notify_rmv() function is used to remove the event notification request. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by stat. The nav_status structure is defined in SPF/nav_api.h.

API specific Errors

EOS_NAV_NOINIT	API not initialized.
EOS_ILLARG	Illegal parameter.

Indirect Errors

Errors generated by _os_setstat().

See Also

nav_notify_rmv()

nav_notify_rmv()

Remove a Request for a Notification on an Asynchronous Event

Syntax

```
#include <SPF/nav_api.h>

error_code nav_notify_rmv
(
   nav_notify_event    event
);
```

Description

Removes a request for notification upon a specific asynchronous event.

Parameters

`event`	A `nav_notify_event` enumerated type specifying the asynchronous event for which the application wishes to remove a request for notification. Currently the only valid values for this parameter are `NAV_NTFY_ON_MAP_MODIFIED` and `NAV_NTFY_ON_CURR_EVENT_AVAILABLE`. The `nav_notify_event` enumerated type is defined in `SPF/nav_api.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized
`EOS_ILLARG`	Illegal parameter

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`nav_notify_asgn()`

nav_remove_chan()

Remove a Channel from a Favorite Channel Ring

Syntax

```
#include <SPF/nav_api.h>

error_code nav_remove_chan
(
   u_int32              ring_id,
   channel              *chan
);
```

Description

Removes the channel whose `channel` structure is pointed to by `chan` from a favorite channel ring. The favorite channel ring is specified by its ring ID.

Parameters

`ring_id`	The ring ID of an existing favorite channel ring. The ring ID is returned as an output parameter from the `nav_create_ring()` call.
`chan`	A pointer to `channel` structure for the channel to be removed from the specified ring.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_NERING`	Non-existent ring.
`EOS_NAV_NECHAN`	Non-existent channel.
`EOS_NAV_INUSE`	The specified channel is the current channel.

Indirect Errors

Errors generated by `_os_getstat()` and `_os_setstat()`.

See Also

`nav_add_chan()`
`nav_list_chans()`
`nav_get_num_chans()`

nav_remove_digital_freq()

Remove a Digital Frequency from the Channel Map

Syntax

```
#include <SPF/nav_api.h>

error_code nav_remove_digital_freq
(
   u_int32              freq
);
```

Description

Removes a digital frequency from the list of tunable frequencies and deletes all channels on this frequency from each channel ring in the channel map.

Parameters

`freq`	The channel's frequency, specified in Hertz for ATSC and Khz for DVB. This frequency should be the same as that contained in a `tuner_pb` structure whose address was passed in a previous `nav_add_digital_freq()` call to add the digital frequency to the channel map.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_add_digital_freq()`
`nav_destroy_analog_chan()`

nav_remove_from_epg_cache()

Remove specified events from the event cache

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_remove_from_epg_cache
(
    u_int32             cache_handle,
    u_int32             ring_id,
    u_int32             major_channel_num,
    u_int32             minor_channel_num,
    channel_option      channel_rmv_option,
    time_t              event_time,
    u_int32             duration_in_seconds,
    time_span_option time_span_rmv_option
);
```

Description

Remove specified events from the EPG cache. The events to be deleted from the cache are selected based on channel and time. This call can used to delete stale events (events in the past) from the EPG cache in order to add new events using the `nav_add_to_epg_cache()`.

Parameters

`cache_handle` Handle to the specified EPG cache.

`ring_id` Id of the channel ring containing the channels from which events are selected for deletion.

`major_channel_number`

Major channel number of the channel from which events are selected for deletion. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.

`minor_channel_number`

The minor channel number of the channel from which events are selected for deletion. For the ATSC Channel Manager, in the US implementation, the minor channel number for an analog channel must be 0. In DVB environments, the minor channel number is the same as the service id of the channel.

`channel_rmv_option`

Enumerated parameter which can take the values `CHANNELS_ALL` or `CHANNEL_SPECIFIED.` If the `CHANNELS_ALL` value is set then events from all channels within the specified time span are removed from the cache. The major and minor channel number values passed in are ignored. If the `CHANNELS_SPECIFIED` value is set, then only events from the specified channel that lie in the time span indicated are removed from the cache.

`event_time` This field indicates the time span used for selecting events for removal from the cache. The interpretation of this field depends on the `time_span_rmv_option`.

`duration_in_seconds`

Length of time (in seconds) after the event_time of the events to be removed. NOTE: This field is only used when `time_span_rmv_option` is set to `TIME_SPAN_SPECIFIED`.

`time_span_rmv_option`

Enumerated parameter used to determine which events to remove with respect to `event_time`. Valid values are:

`TIME_SPAN_ALL`	No time restriction. The `event_time` and `duration` fields are ignored
`TIME_SPAN_BEFORE`	Events ending before the specified `event_time`.
`TIME_SPAN_AFTER`	Events starting after the specified `event_time`.
`TIME_SPAN_SPECIFIED`	All events active between `event_time` and (`event_time` + `duration_in_seconds`).

API specific Errors

`EOS_NAV_NOINIT`	API not initialized
`EOS_ILLARG`	Invalid argument passed to function

Inderect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_add_to_epg_cache()`

`nav_cancel_epg_retrieval_request()`

nav_set_chan_map()

Pass an Initial Channel Map to the Channel Manager

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_chan_map
(
   void                    *chan_map,
   u_int32                 size
);
```

Description

Initializes the Channel Manager Driver's channel map data structures with a channel map passed from the application. The channel map passed from the application should be one which was copied from the Channel Manager before a previous power down cycle.

Before powering down, an application can call `nav_get_ptr_chan_map()` to retrieve a pointer to the channel map and the size of the channel map data. The application can then copy the channel map to non-volatile memory. When the set-top or television is powered back up, the application can retrieve the channel map from non-volatile memory and pass it into the Channel Manager by issuing the `nav_set_chan_map()` call.

The `nav_set_chan_map()` call can only be successfully issued after the Channel Manager has been initialized (via the `nav_init()` call), but before the Channel Manager has created a channel map. The Channel Manager will create a channel map in response to a successful call to any of the following Navigation API functions: `nav_add_digital_freq()`, `nav_create_analog_chan()`, `nav_load_chan_map()`, or `nav_set_chan_map()`.

Parameters

`chan_map`	A `void` pointer which points to the channel map to be copied from the application to the Channel Manager.

size	A u_int32 containing the size (in bytes) of the channel map data.

API specific Errors

EOS_NAV_NOINIT	API not initialized.
EOS_ILLARG	Illegal parameter.

Indirect Errors

Errors generated by _os_setstat().

See Also

nav_add_digital_freq()
nav_create_analog_chan()
nav_get_ptr_chan_map()
nav_load_chan_map()

nav_set_configuration()

Set the Channel Manager's Configuration Settings

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_configuration
(
   nav_config           *config
);
```

Description

Returns the current values of the Channel Manager's configuration settings.

Parameters

`config`	A pointer to a `nav_config` structure. The current configuration settings are returned in `*config`. The `nav_config` structure is defined in `SPF/nav_api.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_set_configuration()`

nav_set_cur_chan_next()

Change Channels to the Next Channel in the Current Channel Ring

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_cur_chan_next
(
  nav_notify          *ntfy,
  nav_status          *stat
);
```

Description

Sets the current channel to the next channel in the current channel ring, causing the channel's audio and video streams to begin playing. The original channel becomes the channel map's previously active channel as well as the current channel ring's previously active channel.

Note

For the ATSC and DVB Channel Manager, if the current channel ring is the main channel ring this call will dynamically determine the next channel number that is currently active and select that channel for viewing. The channel selection will be made from among all frequencies which have been added to the channel map.

If the current channel ring is a favorite channel ring, this call will select the next channel in the current ring, whether that channel is currently active or not.

In both cases the current channel number will be used as the starting point for the selection of the next channel, and the channel selection algorithm will wrap from the last channel number to the first channel number if necessary.

The `nav_set_cur_chan_next()` call can execute either synchronously or asynchronously. The call executes asynchronously when a non-null pointer to a `nav_notify` structure is passed for the `ntfy` parameter. When the `ntfy` parameter is `NULL`, the call executes synchronously, blocking until the next channel is acquired or an error has occurred. In this case the `stat` parameter is ignored.

When the `ntfy` parameter is not `NULL`, it must point to a `nav_notify` structure that specifies the type of asynchronous notification to be issued to the caller when the request has completed. In this case the `stat` parameter must point to a `nav_status` structure. The final status of the operation will be written into this structure before the requested notification is issued. As with all asynchronous requests to the Navigation API, if the return value of the call is `SUCCESS`, the requested notification will eventually be issued and the `nav_status` structure pointed to by the `stat` parameter will contain the final status of the request. However, if the call returns an immediate error code, the requested asynchronous notification will never be issued.

Parameters

`ntfy`	A pointer to a `nav_notify` structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the channel change request has completed. When the `ntfy` parameter is `NULL`, the call is executed synchronously. The `nav_notify` structure is defined in `SPF/nav_api.h`.
`stat`	A pointer to a `nav_status` structure. A status is written to `*stat` once the requested channel change has completed. If the call to `nav_set_cur_chan_next()`

returns SUCCESS, the nav_status structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by stat. The nav_status structure is defined in SPF/nav_api.h.

API specific Errors

EOS_NAV_NOINIT	API not initialized.
EOS_ILLARG	Illegal parameter.
EOS_NAV_NOCURRENT	No current channel ring.
EOS_NAV_NECHAN	Non-existent channel.

Indirect Errors

Errors generated by _os_getstat() and _os_setstat().

See Also

nav_abort()
nav_set_cur_chan_num()
nav_set_cur_chan_prec()
nav_set_cur_chan_prev_active()
nav_set_cur_chan_ptr()
nav_set_cur_chan_ring_prev_active()
nav_set_cur_ring()

nav_set_cur_chan_num()

Change Channels to a Specified Channel Number

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_cur_chan_num
(
   u_int32             major_ch_num,
   u_int32             minor_ch_num,
   nav_notify          *ntfy,
   nav_status          *stat
);
```

Description

Sets the current channel to the specified channel number, causing the channel's audio and video streams to begin playing. The original channel becomes the channel map's previously active channel as well as the current channel ring's previously active channel.

It is not required that the specified channel number correspond to a channel in the current channel ring.

Note

For the ATSC Channel Manager, if an analog channel number is specified, the requested channel will be tuned to even if that channel has not been added to the channel map with the `nav_create_analog_chan()` call. This allows a viewer to select NTSC channels 3 or 4 in order to receive the RF modulated input signal from a VCR or video game. However, analog channels which have not been explicitly added to the channel map will not be selected by the `nav_set_cur_chan_next()` and `nav_set_cur_chan_prec()` calls.

ATSC analog channels are those channels for which the minor channel number is zero.

The nav_set_cur_chan_num() call can execute either synchronously or asynchronously. The call executes asynchronously when a non-null pointer to a nav_notify structure is passed for the ntfy parameter. When the ntfy parameter is NULL, the call executes synchronously, blocking until the next channel is acquired or an error has occurred. In this case the stat parameter is ignored.

When the ntfy parameter is not NULL, it must point to a nav_notify structure that specifies the type of asynchronous notification to be issued to the caller when the request has completed. In this case the stat parameter must point to a nav_status structure. The final status of the operation will be written into this structure before the requested notification is issued. As with all asynchronous requests to the Navigation API, if the return value of the call is SUCCESS, the requested notification will eventually be issued and the nav_status structure pointed to by the stat parameter will contain the final status of the request. However, if the call returns an immediate error code, the requested asynchronous notification will never be issued.

Parameters

major_ch_num	The requested major channel number. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.
minor_ch_num	The requested minor channel number. In DVB environments, the minor channel number is the same as the service id of the channel.
ntfy	A pointer to a nav_notify structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the channel change request has completed. When the ntfy parameter is

	NULL, the call is executed synchronously. The nav_notify structure is defined in SPF/nav_api.h.
stat	A pointer to a nav_status structure. A status is written to *stat once the requested channel change has completed. If the call to nav_set_cur_chan_num() returns SUCCESS, the nav_status structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by stat. The nav_status structure is defined in SPF/nav_api.h.

API specific Errors

EOS_NAV_NOINIT	API not initialized.
EOS_ILLARG	Illegal parameter.
EOS_NAV_NOCURRENT	No current channel ring.
EOS_NAV_NECHAN	Non-existent channel.

Indirect Errors

Errors generated by _os_getstat() and _os_setstat().

See Also

nav_abort()
nav_set_cur_chan_next()
nav_set_cur_chan_prec()
nav_set_cur_chan_prev_active()
nav_set_cur_chan_ptr()
nav_set_cur_chan_ring_prev_active()
nav_set_cur_ring()

nav_set_cur_chan_prec()

Change Channels to the Preceding Channel in the Current Channel Ring

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_cur_chan_prec
(
   nav_notify          *ntfy,
   nav_status          *stat
);
```

Description

Sets the current channel to the preceding channel in the current channel ring, causing the channel's audio and video streams to begin playing. The original channel becomes the channel map's previously active channel as well as the current channel ring's previously active channel.

Note

For the ATSC and DVB Channel Manager, if the current channel ring is the main channel ring this call will dynamically determine the preceding channel number that is currently active and select that channel for viewing. The channel selection will be made from among all frequencies which have been added to the channel map.

If the current channel ring is a favorite channel ring, this call will select the preceding channel in the current ring, whether that channel is currently active or not.

In both cases the current channel number will be used as the starting point for the selection of the preceding channel, and the channel selection algorithm will wrap from the first channel number to the last channel number if necessary.

The `nav_set_cur_chan_prec()` call can execute either synchronously or asynchronously. The call executes asynchronously when a non-null pointer to a `nav_notify` structure is passed for the `ntfy` parameter. When the `ntfy` parameter is `NULL`, the call executes synchronously, blocking until the preceding channel is acquired or an error has occurred. In this case the `stat` parameter is ignored.

When the `ntfy` parameter is not `NULL`, it must point to a `nav_notify` structure that specifies the type of asynchronous notification to be issued to the caller when the request has completed. In this case the `stat` parameter must point to a `nav_status` structure. The final status of the operation will be written into this structure before the requested notification is issued. As with all asynchronous requests to the Navigation API, if the return value of the call is `SUCCESS`, the requested notification will eventually be issued and the `nav_status` structure pointed to by the `stat` parameter will contain the final status of the request. However, if the call returns an immediate error code, the requested asynchronous notification will never be issued.

Parameters

`ntfy`	A pointer to a `nav_notify` structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the channel change request has completed. When the `ntfy` parameter is `NULL`, the call is executed synchronously. The `nav_notify` structure is defined in `SPF/nav_api.h`.
`stat`	A pointer to a `nav_status` structure. A status is written to `*stat` once the requested channel change has completed. If the call to `nav_set_cur_chan_prec()`

returns SUCCESS, the nav_status structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by stat. The nav_status structure is defined in SPF/nav_api.h.

API specific Errors

EOS_NAV_NOINIT	API not initialized.
EOS_ILLARG	Illegal parameter.
EOS_NAV_NOCURRENT	No current channel ring.
EOS_NAV_NECHAN	Non-existent channel.

Indirect Errors

Errors generated by _os_getstat() and _os_setstat().

See Also

nav_abort()
nav_set_cur_chan_next()
nav_set_cur_chan_num()
nav_set_cur_chan_prev_active()
nav_set_cur_chan_ptr()
nav_set_cur_chan_ring_prev_active()
nav_set_cur_ring()

nav_set_cur_chan_prev_active()

Change Channels to the Channel Map's Previously Active Channel

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_cur_chan_prev_active
(
   nav_notify          *ntfy,
   nav_status          *stat
);
```

Description

Sets the current channel to the channel map's previously active channel, causing the channel's audio and video streams to begin playing. The original channel becomes the channel map's previously active channel as well as the current channel ring's previously active channel.

Note

If the previously active channel is in a channel ring that is different than the current channel ring, this function will switch to the previously active channel, but the current channel ring will not be changed.

The `nav_set_cur_chan_ring_prev_active()` API call can be used to switch the current channel ring's previously active channel.

The `nav_set_cur_chan_prev_active()` call can execute either synchronously or asynchronously. The call executes asynchronously when a non-null pointer to a `nav_notify` structure is passed for the `ntfy` parameter. When the `ntfy` parameter is `NULL`, the call executes synchronously, blocking until the previously active channel is acquired or an error has occurred. In this case the `stat` parameter is ignored.

When the ntfy parameter is not NULL, it must point to a nav_notify structure that specifies the type of asynchronous notification to be issued to the caller when the request has completed. In this case the stat parameter must point to a nav_status structure. The final status of the operation will be written into this structure before the requested notification is issued. As with all asynchronous requests to the Navigation API, if the return value of the call is SUCCESS, the requested notification will eventually be issued and the nav_status structure pointed to by the stat parameter will contain the final status of the request. However, if the call returns an immediate error code, the requested asynchronous notification will never be issued.

Parameters

ntfy	A pointer to a nav_notify structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the channel change request has completed. When the ntfy parameter is NULL, the call is executed synchronously. The nav_notify structure is defined in SPF/nav_api.h.
stat	A pointer to a nav_status structure. A status is written to *stat once the requested channel change has completed. If the call to nav_set_cur_chan_prev_active() returns SUCCESS, the nav_status structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by stat. The nav_status structure is defined in SPF/nav_api.h.

API specific Errors

EOS_NAV_NOINIT	API not initialized.

`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_NOCURRENT`	No current channel ring.
`EOS_NAV_NECHAN`	Non-existent channel.

Indirect Errors

Errors generated by `_os_getstat()` and `_os_setstat()`.

See Also

`nav_abort()`
`nav_set_cur_chan_next()`
`nav_set_cur_chan_num()`
`nav_set_cur_chan_prec()`
`nav_set_cur_chan_ptr()`
`nav_set_cur_chan_ring_prev_active()`
`nav_set_cur_ring()`

nav_set_cur_chan_ptr()

Change Channels to a Channel Specified by a Pointer to Its channel Structure

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_cur_chan_ptr
(
    channel             *chan,
    nav_notify          *ntfy,
    nav_status          *stat
);
```

Description

Sets the current channel by a channel pointer within the current channel ring, causing the channel's audio and video streams to begin playing. The original channel becomes the channel map's previously active channel as well as the current channel ring's previously active channel.

The `nav_set_cur_chan_ptr()` call can execute either synchronously or asynchronously. The call executes asynchronously when a non-null pointer to a `nav_notify` structure is passed for the `ntfy` parameter. When the `ntfy` parameter is `NULL`, the call executes synchronously, blocking until the previously active channel is acquired or an error has occurred. In this case the `stat` parameter is ignored.

When the `ntfy` parameter is not `NULL`, it must point to a `nav_notify` structure that specifies the type of asynchronous notification to be issued to the caller when the request has completed. In this case the `stat` parameter must point to a `nav_status` structure. The final status of the operation will be written into this structure before the requested notification is issued. As with all asynchronous requests to the Navigation API, if the return value of the call is `SUCCESS`, the requested notification will eventually be issued and the `nav_status` structure pointed to by the `stat` parameter will contain the final status of the request. However, if the call returns an immediate error code, the requested asynchronous notification will never be issued.

Parameters

`chan`	A pointer to the `channel` structure for the desired channel. The `channel` structure is defined in `SPF/nav_api.h`.
`ntfy`	A pointer to a `nav_notify` structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the channel change request has completed. When the `ntfy` parameter is `NULL`, the call is executed synchronously. The `nav_notify` structure is defined in `SPF/nav_api.h`.
`stat`	A pointer to a `nav_status` structure. A status is written to `*stat` once the requested channel change has completed. If the call to `nav_set_cur_chan_ptr()` returns `SUCCESS`, the `nav_status` structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by `stat`. The `nav_status` structure is defined in `SPF/nav_api.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_NOCURRENT`	No current channel ring.
`EOS_NAV_NECHAN`	Non-existent channel.

Indirect Errors

Errors generated by `_os_getstat()` and `_os_setstat()`.

See Also

nav_abort()
nav_set_cur_chan_next()
nav_set_cur_chan_num()
nav_set_cur_chan_prec()
nav_set_cur_chan_prev_active()
nav_set_cur_chan_ring_prev_active()
nav_set_cur_ring()

nav_set_cur_chan_ring_prev_active

Change Channels to the Current Channel Ring's Previously Active Channel

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_cur_chan_prev_active
(
  nav_notify          *ntfy,
  nav_status          *stat
);
```

Description

Sets the current channel to the current channel ring's previously active channel, causing the channel's audio and video streams to begin playing. The original channel becomes the channel map's previously active channel as well as the current channel ring's previously active channel.

The `nav_set_cur_chan_ring_prev_active()` call can execute either synchronously or asynchronously. The call executes asynchronously when a non-null pointer to a `nav_notify` structure is passed for the `ntfy` parameter. When the `ntfy` parameter is `NULL`, the call executes synchronously, blocking until the previously active channel is acquired or an error has occurred. In this case the `stat` parameter is ignored.

When the `ntfy` parameter is not `NULL`, it must point to a `nav_notify` structure that specifies the type of asynchronous notification to be issued to the caller when the request has completed. In this case the `stat` parameter must point to a `nav_status` structure. The final status of the operation will be written into this structure before the requested notification is issued. As with all asynchronous requests to the Navigation API, if the return value of the call is `SUCCESS`, the requested notification will eventually be issued and the `nav_status` structure pointed to by the `stat` parameter will contain the final status of the request. However, if the call returns an immediate error code, the requested asynchronous notification will never be issued.

Parameters

ntfy	A pointer to a nav_notify structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the channel change request has completed. When the ntfy parameter is NULL, the call is executed synchronously. The nav_notify structure is defined in SPF/nav_api.h.
stat	A pointer to a nav_status structure. A status is written to *stat once the requested channel change has completed. If the call to nav_set_cur_chan_prev_active() returns SUCCESS, the nav_status structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by stat. The nav_status structure is defined in SPF/nav_api.h.

API specific Errors

EOS_NAV_NOINIT	API not initialized.
EOS_ILLARG	Illegal parameter.
EOS_NAV_NOCURRENT	No current channel ring.
EOS_NAV_NECHAN	Non-existent channel.

Indirect Errors

Errors generated by _os_getstat() and _os_setstat().

See Also

nav_abort()

```
nav_set_cur_chan_next()
nav_set_cur_chan_num()
nav_set_cur_chan_prec()
nav_set_cur_chan_prev_active()
nav_set_cur_chan_ptr()
nav_set_cur_ring()
```

nav_set_cur_ring()

Change the Current Channel Ring

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_cur_ring
(
    u_int32             ring_id,
    u_int32             major_ch_num,
    u_int32             minor_ch_num,
    nav_notify          *ntfy,
    nav_status          *stat
);
```

Description

Changes the current channel ring and changes the current channel to a channel within the specified channel ring. The `major_ch_num` and `minor_ch_num` parameters specifies the new channel within the specified channel ring. This parameter may be passed as zero to change to the ring's current channel. (The ring's current channel will be the channel that was currently selected at the last time the specified channel ring was the current channel ring.)

The `nav_set_cur_ring()` call can execute either synchronously or asynchronously. The call executes asynchronously when a non-null pointer to a `nav_notify` structure is passed for the `ntfy` parameter. When the `ntfy` parameter is `NULL`, the call executes synchronously, blocking until the previously active channel is acquired or an error has occurred. In this case the `stat` parameter is ignored.

When the `ntfy` parameter is not `NULL`, it must point to a `nav_notify` structure that specifies the type of asynchronous notification to be issued to the caller when the request has completed. In this case the `stat` parameter must point to a `nav_status` structure. The final status of the operation will be written into this structure before the requested notification is issued. As with all asynchronous requests to the Navigation API, if the return value of the call is `SUCCESS`, the requested notification will eventually be issued and the `nav_status` structure pointed to by the `stat` parameter will contain the final status of the request. However, if the call returns an immediate error code, the requested asynchronous notification will never be issued.

Parameters

`ring_id`	The ring ID of the ring to become the new current channel ring. For favorite channel rings the ring ID is returned as an output parameter from the `nav_create_ring()` call. The ring ID of the main channel ring is zero.
`major_ch_num`	The requested major channel number. If a value of zero is passed for `major_ch_num and minor_ch_num`, the ring's current channel will be selected. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.
`minor_ch_num`	The requested minor channel number. If a value of zero is passed for `major_ch_num and minor_ch_num`, the ring's current channel will be selected. In DVB environments, the minor channel number is the same as the service id of the channel.
`ntfy`	A pointer to a `nav_notify` structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the channel change request has completed. When the `ntfy` parameter is `NULL`, the call is executed synchronously. The `nav_notify` structure is defined in `SPF/nav_api.h`.
`stat`	A pointer to a `nav_status` structure. A status is written to `*stat` once the requested channel change has completed.

If the call to nav_set_cur_ring() returns SUCCESS, the nav_status structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by stat. The nav_status structure is defined in SPF/nav_api.h.

API specific Errors

EOS_NAV_NOINIT	API not initialized.
EOS_ILLARG	Illegal parameter.
EOS_NAV_NERING	Non-existent ring.
EOS_NAV_NOCURRENT	No current channel.
EOS_NAV_NECHAN	Non-existent channel.

Indirect Errors

Errors generated by _os_getstat() and _os_setstat().

See Also

nav_abort()
nav_set_cur_chan_next()
nav_set_cur_chan_num()
nav_set_cur_chan_prec()
nav_set_cur_chan_prev_active()
nav_set_cur_chan_ptr()
nav_set_cur_chan_ring_prev_active()

nav_set_preferences()

Set the Channel Manager's User Preference Settings

Syntax

```
#include <SPF/nav_api.h>

error_code nav_get_preferences
(
  ch_user_pref        *prefs
);
```

Description

Sets the user preferences (such as `language` preferences for audio track selection) used by the Channel Manager Device Driver.

Parameters

`prefs`	A pointer to a `ch_user_pref` structure containing the desired Channel Manager user preferences. The `ch_user_pref` structure is defined in `SPF/nav_api.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_get_preferences()`

nav_set_ring_name()

Set the Name of a Favorite Channel Ring

Syntax

```
#include <SPF/nav_api.h>

error_code nav_set_ring_name
(
    u_int32                  ring_id,
    char                     *ring_name
);
```

Description

Changes the name of an existing favorite channel ring (specified by `ring_id`) to the name pointed to by `ring_name`.

Parameters

`ring_id`	The ring ID of an existing favorite channel ring. The ring ID is returned as an output parameter from the `nav_create_ring()` call.
`ring_name`	A pointer to a character string specifying the new name for the channel ring. The string should be `NULL` terminated and the length of the string should be less than or equal to `RING_MAX_NAME` characters, including the `NULL` byte. The `RING_MAX_NAME` macro is defined in `SPF/nav_api.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.
`EOS_NAV_AERING`	Duplicate ring name.
`EOS_NAV_NERING`	Non-existent ring.

`EOS_NAV_NOTALLOWED`	The specified ring is not a favorite channel ring.

Indirect Errors

Errors generated by `_os_getstat()` and `_os_setstat()`.

See Also

`nav_clear_chan_map()`
`nav_delete_chan_map_file()`

nav_setstat()

Pass an OEM Defined Setstat to a DBE Hardware Driver

Syntax

```
#include <SPF/nav_api.h>

error_code nav_setstat
(
   void                *pb
);
```

Description

Passes an OEM defined setstat request to the Tuner Driver or Conditional Access Driver. The Channel Manager will route the setstat request to the proper DBE driver.

The pb parameter must point to a memory area which contains a valid spf_ss_pb structure at the start of the memory. The Channel Manager Driver will use the protocol ID value in the upper 16 bits of the code field of this spf_ss_pb structure to route the setstat request to the proper DBE driver. The recommended method for using this function is to define a structure whose first field is an spf_ss_pb structure and pass the address to this enclosing structure as the pb parameter. The enclosing structure can contain the additional fields necessary to support the OEM defined setstat request.

The bottom 16 bits of the code field of the spf_ss_pb structure define the specific setstat request being issued. In order to prevent conflicts with setstat codes defined by the DBE package, these bits should contain a value of hex 100 or greater.

Note

The Channel Manager Driver contains a variable that records the currently tuned frequency. In order to maintain the consistency of this variable, OEM defined setstat requests implemented by the Tuner Driver should not result in a frequency change. To accomplish an explicit frequency change, an application should instead issue the `nav_tune()` call.

Parameters

pb	A pointer to the getstat parameter block. The getstat parameter block memory should contain an `spf_ss_pb` structure as its first field. The `spf_ss_pb` structure is defined in `SPF/spf.h`.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_getstat()`
`nav_tune()`

nav_term()

Terminate Use of the Navigation API

Syntax

```
#include <SPF/nav_api.h>

error_code nav_term(void);
```

Description

Terminates use of the Navigation API, including closing the path (opened by `nav_init()`) to the Channel Manager Device Driver. Each process which uses the Navigation API should call `nav_term` when it wishes to terminate use of the API.

Parameters

None.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_close()`

See Also

`nav_init()`

nav_tune()

Tune to a Specified Frequency

Syntax

```
#include <SPF/nav_api.h>
#include <SPF/tuner.h>

error_code nav_tune
(
   tuner_pb             *ptr_tpb,
   nav_notify           *ntfy,
   nav_status           *stat,
   u_int32              requested_si_flags
);
```

Description

Tunes to specified frequency. The tuning parameters are specified in a `tuner_pb` structure that the `ptr_tpb` parameter points to.

The `nav_tune()` call can execute either synchronously or asynchronously. The call executes asynchronously when a non-null pointer to a `nav_notify` structure is passed for the `ntfy` parameter. When the `ntfy` parameter is `NULL`, the call executes synchronously, blocking until the specified frequency has been acquired by the Tuner Driver or an error has occurred. In this case the `stat` parameter is ignored.

When the `ntfy` parameter is not `NULL`, it must point to a `nav_notify` structure that specifies the type of asynchronous notification to be issued to the caller when the tune request has completed. In this case the `stat` parameter must point to a `nav_status` structure. The final status of the tuning operation will be written into this structure before the requested notification is issued. As with all asynchronous requests to the Navigation API, if the return value of the call is `SUCCESS`, the requested notification will eventually be issued and the `nav_status` structure pointed to by the `stat` parameter will contain the final status of the request. However, if the call returns an immediate error code, the requested asynchronous notification will never be issued.

An application can use the nav_tune() call to tune to each possible frequency during a frequency scan procedure in order to build a channel map containing data for each valid frequency. This is discussed further in Chapter 2.

Note

During normal channel change operations the Channel Manager will tune to the correct frequency as part of the channel change request. Thus, during channel changes the application does not need to issue an explicit nav_tune() call.

Parameters

ptr_tpb	A pointer to a tuner_pb structure containing the tuning parameters. This structure specifies the requested frequency and whether the frequency is analog or digital. The tuner_pb structure is defined in SPF/tuner.h.
ntfy	A pointer to a nav_notify structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the tune operation has completed. When the ntfy parameter is NULL, the call is executed synchronously. The nav_notify structure is defined in SPF/nav_api.h.
stat	A pointer to a nav_status structure. A status is written to *stat once the tune request has completed. If the call to nav_tune() returns SUCCESS, the nav_status structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will

	not be issued and the Channel Manager will not write to the address pointed to by `stat`. The `nav_status` structure is defined in `SPF/nav_api.h`.
`requested_si_flags`	
	If this flag is zero then the Channel Manager will send the tune complete notification immediately after tuning to the specified frequency. It will not wait for any SI information to be collected. If the flag is set to `SI_CHANNEL_LIST`, then the Channel Manager will wait until the SI channel information has been collected before sending the tune complete notification. If the flag is set to `SI_CURRENT_TIME`, then the Channel Manager will wait until the current time has been obtained from the network before sending the tune complete notification.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`nav_add_digital_freq()`
`nav_create_analog_chan()`
`nav_get_signal_info()`

nav_unlock_map()

Unlock the Channel Map

Syntax

```
#include <SPF/nav_api.h>

error_code nav_unlock_map(void);
```

Description

Unlocks the channel map to allow updates. A `nav_unlock_map()` call should be issued for each `nav_lock_map()` call issued by an application.

Parameters

None.

API specific Errors

`EOS_NAV_NOINIT`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`nav_lock_map()`

nav_update_minor_channel_list()

Have the Channel Manager update list of minor channels for a specified major channel

Syntax

```
#include <SPF/epg_api.h>
#include <SPF/nav_api.h>

error_code nav_update_minor_channel_list
(
   u_int32             ring_id,
   u_int32             major_channel_num,
   nav_notify          *ntfy,
   nav_status          *stat
);
```

Description

This call forces the channel manager to update the list of minor channels for a given major channel number.

Note

This call can be used to update the channel map for the specified major channel number before nav_list_preceding_minor_channels(), and nav_list_succeeding_minor_channels() are called.

Parameters

`ring_id`	Id of the main ring containing the specified major channel number.
`major_channel_num`	Specified major channel number. For DVB, the major channel number uniquely identifies the frequency on which the channel resides. It contains

the original network id of the channel in the 16 MSB and the transport id of the channel in the 16 LSB.

`ntfy`	A pointer to a `nav_notify` structure. This structure specifies the method of notification (signal or event) to be used to notify the caller when the tune operation has completed. When the `ntfy` parameter is `NULL`, the call is executed synchronously. The `nav_notify` structure is defined in `SPF/nav_api.h`.
`stat`	A pointer to a `nav_status` structure. A status is written to `*stat` once the request has completed. If the call returns `SUCCESS`, the `nav_status` structure must remain allocated by the application until the requested notification is received. If an error is returned, the requested notification will not be issued and the Channel Manager will not write to the address pointed to by `stat`. The `nav_status` structure is defined in `SPF/nav_api.h`.

API specific Errors

`EOS_NAV_NOINT`	API not initialized.
`EOS_ILLARG`	Invalid argument passed to function.
`EOS_NAV_NERING`	Specified ring not found.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

```
nav_list_preceding_major_channels()
nav_list_preceding_minor_channels()
nav_list_succeeding_major_channels()
nav_list_succeeding_minor_channels()
```

Chapter 4: The MPEG Private Data API

This chapter contains descriptions, in alphabetical order, of the MPEG Private Data API functions.

Function Descriptions

The function descriptions are, for the most part, self-explanatory. Each function description contains the following sections:

The SYNTAX section shows the function prototype with the required parameters and their data types.

The DESCRIPTION section provides a description of the function.

The PARAMETERS section provides details about each of the function's parameters.

FATAL ERRORS are errors detected within the API call and are returned directly by that particular call. Applications may not be able to recover from fatal errors.

NON-FATAL ERRORS are errors detected within the API call and are a direct result of that particular call. Applications can recover from non-fatal errors.

INDIRECT ERRORS are errors returned by another function called during the processing of the API request.

The SEE ALSO section lists related functions or materials that provide more information about the function.

MPEG Private Data Functions

The table below summarizes the MPEG Private Data functions:

Table 4-1 Summary of MPEG Private Data Functions

Function	Description
`mpg_block_type()`	Set read blocking behavior.
`mpg_chg_table_mask()`	Change the section mask.
`mpg_count_sections()`	Count the number of sections available to be read.
`mpg_data_avail_asgn()`	Send notification on data available.
`mpg_data_avail_rmv()`	Remove notification request.
`mpg_data_notify_asgn()`	Send a notification when data is ready.
`mpg_data_notify_rmv()`	Remove data notification.
`mpg_flush_sections()`	Flush table sections.
`mpg_free_mbuf()`	Frees an mbuf obtained by `mpg_read_section_mbufs()`
`mpg_init()`	Initialize MPEG Private Data API.
`mpg_read_sections()`	Read sections.
`mpg_read_section_mbufs()`	Read mbufs containing sections.

Table 4-1 Summary of MPEG Private Data Functions (continued)

Function	Description
`mpg_register_table()`	Register for a table.
`mpg_term()`	Terminate MPEG Private Data API
`mpg_unregister_table()`	Unregister for a table.

mpg_block_type()

Set Read Blocking Behavior

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_block_type(u_int16 block_flag);
```

Description

Determines the behavior of `mpg_read_sections()` and `mpg_read_section_mbufs()`. When `block_flag` is `0`, read calls do not block if data is currently not available. Instead, a buffer size of zero or a NULL mbuf queue is returned. When `block_flag` is set to `1`, read calls block and wait until data becomes available before returning.

Note

When the MPEG Private Data API is initialized, read blocking is *off* by default.

Parameters

`block_flag`		Acceptable values are:
	`0`	read blocking should be off
	`1`	read blocking should be on.

Non-Fatal Errors

`EOS_NOTRDY`	API not initalized.
`EOS_ILLPRM`	Illegal parameter.

Indirect Errors

Errors generated by `_os_gs_popt()` and `_os_ss_popt()`.

See Also

```
mpg_read_sections()
mpg_read_section_mbufs()
```

mpg_chg_table_mask()

Change the Section Mask

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_chg_table_mask
(
    u_int32                  table_handle,
    u_char                   new_mask[8],
    u_char                   new_value[8]
);
```

Description

This call changes the mask associated with the reading of a specific table. The change takes effect immediately, as any buffered sections not matching the new mask are discarded.

Parameters

`table_handle`	Handle returned by mpg_register_table().
`new_mask`	New bit array specifying the actual header fields needing to be checked for sections to be selected.
`new_value`	New bit array specifying the values of the header fields for sections to be selected.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_PGM_TBLNFND`	Invalid table handle.
`EOS_ILLPRM`	Illegal parameter.

Indirect Errors

Errors generated by `ite_data_ready()`, `ite_data_readmbuf()`, and `_os_setstat()`

See Also

`mpg_register_table()`
`mpg_flush_sections()`

mpg_count_sections()

Count Sections Available to be Read

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_count_sections
(
    u_int32             table_handle,
    u_int32             *count
);
```

Description

Counts the number of sections waiting to be read. If `table_handle` is zero, all sections are counted. Otherwise, only sections belonging to the specified table are counted.

Parameters

`table_handle`	Handle returned by `mpg_register_table()`, or zero to count all sections in all tables.
`count`	Pointer to a `u_int32` variable where the section count is returned.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLPRM`	Illegal parameter.

Indirect Errors

Errors generated by `ite_data_ready()` and `ite_data_readmbuf()`.

See Also

`mpg_read_sections()`
`mpg_read_section_mbufs()`

mpg_data_avail_asgn()

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_data_avail_asgn(mpg_notify *ntfy);
```

Description

Sets up a notification sent when valid section data for *any* registered table arrives.

Parameters

`ntfy`	Pointer to an `mpg_notify` structure initialized by the application specifying a signal to be sent or an event to be set.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLPRM`	Illegal Parameter.

Indirect Errors

Errors generated by `ite_data_avail_asgn()`.

See Also

`mpg_data_avail_rmv()`

mpg_data_avail_rmv()

Remove Notification Request

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_data_avail_rmv(void);
```

Description

Removes a notification request set up by `mpg_data_avail_asgn()`. This call does not remove notifications set up by `mpg_data_notify_asgn()`.

Parameters

None

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.

Indirect Errors

Errors generated by `ite_data_avail_rmv()`.

See Also

`mpg_data_avail_asgn()`

mpg_data_notify_asgn()

Send a Notification
When Data Ready

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_data_notify_asgn
(
   u_int32               table_handle,
   mpg_notify            *ntfy
);
```

Description

Request a notification when section data is available to be read for a particular table specified by table handle. If data is already enqueued when the call is made, a notification is sent immediately.

Parameters

`table_handle`	Handle returned by `mpg_register_table()`.
`ntfy`	Pointer to an `mpg_notify` structure initialized by the application specifying a signal to be sent or an event to be set.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_PGM_TBLNFND`	Invalid table handle.
`EOS_ILLPRM`	Illegal parameter.
`EOS_NTFY_NFND`	No notification pending for specified table.

Indirect Errors

Errors generated by `_os_setstat()`

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_data_notify_rmv
(
   u_int32              table_handle
);
```

Description

This call removes a registered notification for section data on a particular table specified by `table_handle`. If `table_handle` is zero, and if a notification has been registered for data on any section of any table, then that notification is removed.

Note

Using a `table_handle` of `0` does not remove a notify request made by `mpg_data_avail_asgn()`.

Parameters

`table_handle`	Handle returned by `mpg_register_table()`.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_PGM_TBLNFND`	Invalid table handle.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`mpg_data_notify_asgn()`

mpg_flush_sections()

Flush Table Sections

Syntax

```
#include <SPF/mpg_api.h>

error_code mpgmpg_flush_sections
(
   u_int32              table_handle
);
```

Description

The MPEG Private Data API buffers table sections as they arrive from the Real-Time Device Driver. If an application wishes to flush these sections without reading them, this call should be used. When `table_handle` is `0`, all sections from all tables are flushed.

Parameters

`table_handle`	Handle returned by `mpg_register_table()`, or `0` to flush all sections from all tables.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_PGM_TBLNFND`	Invalid table handle.

Indirect Errors

None.

See Also

`mpg_read_sections()`
`mpg_read_section_mbufs()`
`mpg_chg_table_mask()`
`mpg_unregister_table()`

mpg_free_mbuf()

Free an mbuf Obtained by mpg_read_section_mbufs()

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_free_mbuf
(
   Mbuf                mbuf_ptr
);
```

Description

Frees the mbuf pointed to by `mbuf_ptr`. All mbufs obtained by the `mpg_read_section_mbufs()` call should be freed.

Parameters

`mbuf_ptr`	A pointer to the mbuf to be freed.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLPRM`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`mpg_read_section_mbufs()`

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_init(void);
```

Description

Initializes the MPEG Private Data API and opens a path to the Real-Time Network Driver. This call must be made before any other call in the API can be made.

Parameters

None.

Non-Fatal Errors

EOS_DEVBSY	API already initialized.

Indirect Errors

Errors generated by `ite_path_open()` and `_os_setstat()`

See Also

`mpg_term()`

mpg_read_sections()

Read Sections

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_read_sections
(
  u_int32             table_handle,
  u_int16             payload_flag,
  void                *buffer,
  u_int32             *size
);
```

Description

This call copies SI sections into a buffer passed in by the application. The buffer must be big enough to hold at least one section. The call copies as many sections as it can into the buffer.

If `table_handle` is zero, then any section from any registered table is copied into the buffer, on a first-in first-out (FIFO) basis. If the `table_handle` is set to a specific non-zero value returned by `mpg_register_table()`, then this call gets SI sections matching the mask/value passed in during that `mpg_register_table()`.

The application has the option of reading only the section data (section headers stripped off), or preserving the section headers. When `payload_flag` is `1`, section headers are stripped off. When `payload_flag` is `0`, section headers are preserved.

If no sections are available to be returned by this call and read blocking is off, then a size of zero is returned along with an `EWOULDBLOCK` error code. Use the `mpg_data_notify_asgn()` and `mpg_data_avail_asgn()` calls to get a notification when additional section data is available.

Parameters

`table_handle`		Handle returned by `mpg_register_table()`, or `0` to read sections from any table.
`payload_flag`		Set flag to:
	`0`	table headers are preserved
	`1`	table headers are stripped.
`buffer`		Pointer to user buffer.
`size`		Contains the size of the user buffer on input. The API sets it to hold the size of the actual amount of data in bytes copied into the user buffer.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_PGM_TBLNFND`	Invalid table handle.
`EOS_ILLPRM`	Illegal parameter.
`EBUFTOOSMALL`	Buffer is too small.
`EWOULDBLOCK`	Read operation would normally block, but read blocking is off.

Indirect Errors

Errors generated by `ite_data_ready()` and `ite_data_readmbuf()`.

See Also

`mpg_read_section_mbufs()`
`mpg_chg_table_mask()`
`mpg_block_type()`.

mpg_read_section_mbufs()

Read Mbufs
Containing Sections

Syntax

```
#include <SPF/mpg_api.h>
#include <mbuf.h>

error_code mpg_read_section_mbufs
(
   u_int32              table_handle,
   Mbuf                 *mbq
);
```

Description

This call gets the actual system mbufs containing the SI sections. This saves a copy operation between the system layer and the application layer. The mbufs are queued together using the `m_qnext` field in the mbuf header. The `m_offset` field in the mbuf points to the start of the section and the `m_size` field in the mbuf indicates the size of the section.

If `table_handle` is zero, then all cached mbufs from all registered tables are returned in the mbuf queue passed back to the application. If the `table_handle` is set to a specific non-zero value returned by `mpg_register_table()`, then this call gets only section mbufs matching the mask/value passed in during that `mpg_register_table()`.

If no sections are available to be returned by this call and read blocking is off, then a `NULL` mbuf queue is returned along with an `EWOULDBLOCK` error code. Use the `mpg_data_notify_asgn()` and `mpg_data_avail_asgn()` calls to get a notification when additional data is available.

Note
When an application reads section mbufs, the application is responsible for freeing the mbufs back to the mbuf pool after the read. Be sure to perform an `mpg_free_mbuf()` on each mbuf in the queue or an mbuf leak will result!

Parameters

`table_handle`	Handle returned by `mpg_register_table()`, or `0` to read all mbufs from all tables.
`mbq`	Pointer to queue of mbufs returned to application.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_PGM_TBLNFND`	Invalid table handle.
`EOS_ILLPRM`	Illegal parameter.
`EWOULDBLOCK`	Read operation would normally block, but read blocking is off.

Indirect Errors

Errors generated by `ite_data_ready()` and `ite_data_readmbuf()`.

See Also

`mpg_read_sections()`
`mpg_chg_table_mask()`
`mpg_block_type()`
`mpg_free_mbuf()`

mpg_register_table()

Register For a Table

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_register_table
(
	u_int32			*table_handle,
	u_int16			pid,
	u_char			mask[8],
	u_char			value[8],
	u_int8			flags,
	mpg_notify		*ntfy
);
```

Description

This call registers with the Real-Time Network Driver controlling access to the MPEG-2 demultiplexor for an SI table. The table is specified by the `pid` indicating the elementary stream ID carrying the table and by the 8-byte `mask` and `value` parameters.

Each section of an SI table usually has a 3- or a 8-byte header depending on whether the section syntax indicator is 0 or 1 respectively. The exact format of the SI section is shown below:

Table 4-2 Table section syntax for 8 byte section header

Header fields	Number of bits
table id	8
section_syntax_indicator	1
reserved	3

Table 4-2 Table section syntax for 8 byte section header (continued)

Header fields	Number of bits
`section length`	12
`table_ext_id`	16
`reserved`	2
`version_number`	5
`current_next_indicator`	1
`section_number`	8
`last_section_number`	8
`data`	n
`CRC`	32

`table_handle`	Returned by the API and is used to uniquely identify the call from other `mpg_register_table()` calls. It is passed in to `mpg_read_sections()` and `mpg_abort_table()` as well as other calls.
`pid`	Identifies the PID of the elementary stream carrying the table.
`mask`	Bit array specifying the actual header fields needing to be checked for sections to be selected.
`value`	Bit array specifying the values of the header fields for sections to be selected.

`flags`	If set to `MPGF_TERM_ON_RETUNE`, then the Real-Time driver will automatically terminate the table read operation when the frequency changes.
`mpg_notify`	The Real-Time driver will send a notification to the application when the frequency changes via this notify parameter block.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLPRM`	Illegal parameter.
`EOS_NORAM`	Unable to allocate memory for new table entry.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`mpg_unregister_table()`
`mpg_read_sections()`
`mpg_read_section_mbufs()`
`mpg_chg_table_mask()`

mpg_term()

Terminate Use of the MPEG Private Data API

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_term(void);
```

Description

Terminates use of the MPEG Private Data API, including closing the path to the Real-Time Network Driver and flushing all cached sections.

Parameters

None.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.

Indirect Errors

Errors generated by `ite_path_close()`.

See Also

`mpg_init()`

mpg_unregister_table()

Unregister From a Table

Syntax

```
#include <SPF/mpg_api.h>

error_code mpg_unregister_table
(
    u_int32                 table_handle
);
```

Description

This call tells the system layer software controlling the MPEG-2 demultiplexor chip that the table sections specified by `table_handle` are no longer needed. All cached sections corresponding to the `table_handle` are flushed.

Parameters

`table_handle`	Handle returned by `mpg_register_table()`.

Non-Fatal Errors

`EOS_NOTRDY`	API not initialized.
`EOS_PGM_TBLNFND`	Invalid table handle.

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`mpg_register_table()`

Chapter 5: The DBE Pak Device Driver Architecture

This chapter presents an overview of the device drivers included with the Digital Broadcast Environment Pak. Subsequent chapters describe the individual drivers in detail.

Introduction

The DBE Pak contains four device drivers working in conjunction to perform much of the functionality of the package. These four drivers are the Channel Manager Protocol Driver, Tuner Device Driver, Conditional Access Device Driver, and Real-Time Network Driver. This chapter describes the sequence of interactions between these drivers. Chapters 6 - 9 discuss the functionality of the individual drivers in detail.

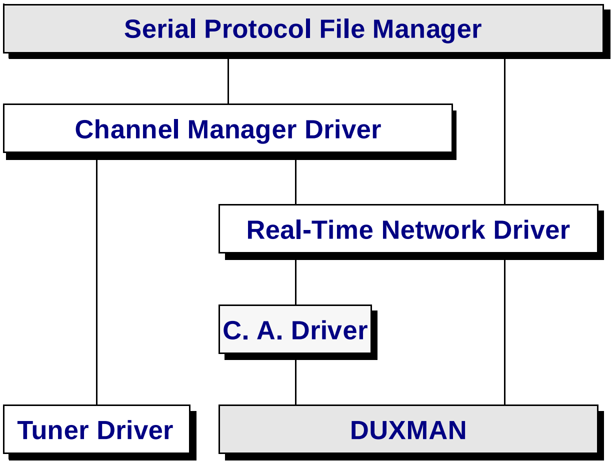
Figure 5-1 DBE Device Driver Architecture shows the architecture of the DBE Pak's device drivers. Each of the four device drivers in the package executes under the SPF File Manager.

As discussed before, the Conditional Access driver is optional.

For More Information

For more information on the structure of SPF device drivers see the ***SPF Porting Guide***.

Figure 5-1 DBE Device Driver Architecture



The Channel Manager Protocol Driver and Real-Time Device Driver are the only drivers directly accessed by applications. An application performing channel change operations contains an open path to the Channel Manager Protocol Driver. The application uses this path to obtain information on available channels and request channel change operations. If the application is using the Navigation API Library, this path is automatically opened for the application by the Navigation API Library's `nav_init()` call.

An application requiring data from the incoming MPEG-2 transport stream contains an open path to the Real-Time Device Driver. The application uses this path to issue requests for the desired data and read the requested data once it has been obtained from the network. If the application is using the MPEG Private Data API Library, this path is automatically opened for the application during the MPEG Private Data API Library's `mpg_init()` call.

The Tuner Device Driver and Conditional Access Device Driver are typically accessed only by the Channel Manager Protocol Driver and the Real-Time Device Driver. The Channel Manager Protocol Driver may issue requests to these drivers when processing a channel change request from an application. The Tuner Device Driver is responsible for tuning to a new frequency when the requested channel is contained within a transport stream other than the one which is currently tuned. The Conditional Access Device Driver is responsible for communicating with the Conditional Access hardware in order to request authorization and decryption when the selected channel is encrypted.

In addition, the Channel Manager Protocol Driver issues calls directly to the Real-Time Device Driver. These calls are used to request tables from the incoming MPEG-2 transport stream and to start and stop play-out of the selected channel.

Although the Channel Manager Protocol Driver makes calls to the Tuner and Real-Time Device Drivers, applications using the Channel Manager Protocol Driver do not need to be aware of the existence of these other drivers. The Channel Manager Protocol Driver opens a dedicated path to each of these drivers during initialization. The existence of these paths is hidden from the application, and these drivers do not appear as part of the SPF protocol stack for the application's path to the Channel Manager Protocol Driver.

Similarly, the Real-Time and Conditional Access Device Drivers open a dedicated path to each other in order to handle conditional access information for the incoming MPEG-2 transport stream. The existence of these paths are also hidden from applications.

Asynchronous Notifications

Many of the requests to the drivers in the DBE Pak can be specified by the caller to operate either synchronously or asynchronously. When specified to operate asynchronously, a call returns control to the calling entity before the requested operation has been completed. For these cases the drivers in the DBE Pak provide a mechanism to let the caller specify the type of notification it wishes to receive when the operation has completed. This notification method can be specified as either an OS-9 signal; an OS-9 event; or, for callers executing in system state, invocation of a call-back function.

The caller specifies the type of request (synchronous or asynchronous) and the notification method for asynchronous calls by passing a pointer to a `notify_type` structure as a parameter to the call.

Note

Once an asynchronous call has been issued, the caller is free to over-write or deallocate the `notify_type` structure. Therefore, driver calls executing asynchronously must copy the necessary information from the `notify_type` structure before returning to the caller.

Asynchronous calls to drivers in the DBE Pak perform some error checking on the passed parameters immediately, but other error conditions can not be detected until the requested operation has been attempted. Thus, some error codes are returned immediately as the return value of the call itself, while other error codes are returned in the `return_type` structure. When the return value from the asynchronous call is anything other than `SUCCESS`, an immediate error was detected and the return value is the error code. In this case, the requested operation was not issued and therefore the asynchronous notification never occurs. When the return value from the asynchronous call is `SUCCESS`, the asynchronous operation has been issued. However, this does not imply the operation necessarily completes successfully. In this case, the requested asynchronous notification occurs and the actual status of the operation is returned in the `return_type` structure.

Note

The `return_type` structure is defined in the header file `SPF/item.h`.

notify_type

Declaration

The `notify_type` structure is declared in the file `SPF/item.h` as follows:

```
typedef struct notify_type
{
   struct notify_type*ntfy_next;
   u_charntfy_class;
   u_charntfy_on;
   u_charntfy_rsv1;
   u_charntfy_ctl_type;
   void*ntfy_ctl;
   u_int32ntfy_timeout;
   u_int32ntfy_rsv[2];

   union
   {
   struct
   {
   u_int32proc_id;
   u_int32sig2send;
   } sig;

   struct
   {
   u_int32ev_id;
   int32ev_val;
   } ev;

   struct
   {
   u_int32ev_id;
   int32ev_inc_val;
   } inc_ev;
```

```
    struct
    {
    u_int32mmbox_handle;
    error_code(*callback_func)();
    } mmbox;

    struct
    {
    void *callbk_param;
    error_code(*callback_func)();
    } callbk;
    } notify;
} notify_type, *Notify_type;
```

Fields

The caller fills in the `ntfy_class`, `ntfy_ctl_type`, `ntfy_ctl`, and `notify` fields of the structure before issuing a call to the driver. The remaining fields are for internal driver use.

The fields set by the caller are described below:

`ntfy_class` specifies whether the request should execute synchronously or asynchronously, and for asynchronous requests the type of notification desired. Valid values are:

- `ITE_NCL_BLOCK` specifies a synchronous request.
- `ITE_NCL_SIGNAL` specifies an asynchronous request with notification via a signal.
- `ITE_NCL_EVENT` specifies an asynchronous request with notification via an event.
- `ITE_NCL_CALLBACK` specifies an asynchronous request with notification via a call-back function.

Note

When the `ITE_NCL_CALLBACK` `ntfy_class` is specified, the caller must be executing in system state. Typically this `ntfy_class` is used only by device drivers issuing asynchronous requests to other device drivers.

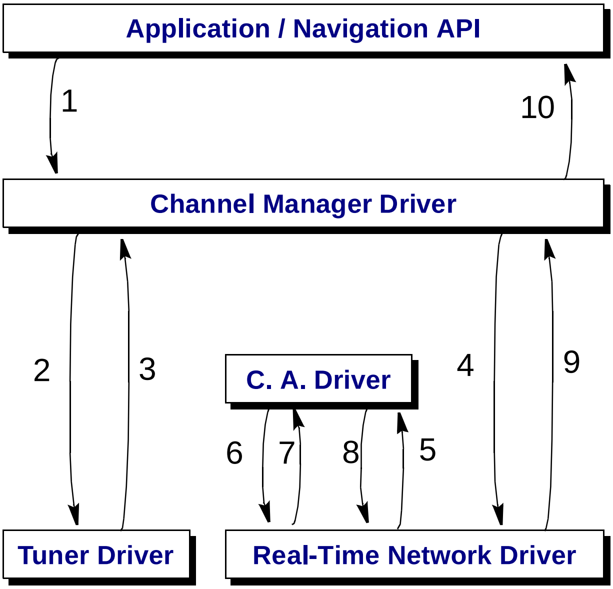
`ntfy_ctl_type`	identifies the type of pointer set in the `ntfy_ctl` field. For asynchronous calls, the `ntfy_ctl_type` field should be set to `NTYPE_RETURN`. For synchronous calls, the `ntfy_ctl_type` field should be set to `NTYPE_NONE`.
`ntfy_ctl`	set to point to the caller's `return_type` structure for asynchronous calls. This structure contains a single field set by the driver to the completion status of the asynchronous call before the notification is issued. The `return_type` structure is defined in the file `SPF/item.h`. The `return_type` structure simply provides a location for an error code to be returned. Thus, this structure must remain allocated and unused by the caller until the asynchronous call has completed. When an caller is notified an asynchronous call has completed, it should check the `return_type` structure to determine whether or not the call was successful.
`proc_id`	is for internal driver use.
`sig2send`	specifies the signal number desired for a notification via a signal.
`ev_id`	specifies the event identifier for a notification via an event.

`ev_value`	specifies the desired event value for a notification via an event.
`callback_func`	specifies the address of the call-back function for a notification via a call-back function, this field .
`callbk_param`	specifies the desired parameter value for the call-back function for a notification via a call-back function.
`notify`	is a union specifying the parameters for the desired asynchronous notifications.

Channel Change Operations

This section describes the flow of control between the different drivers for channel change operations. The figure below illustrates this sequence.

Figure 5-2 Driver Control for Channel Change Operations



The channel change process is as follows:

1. When an application requests a channel change via a call to the Navigation API, the Navigation API forwards the channel change request to the Channel Manager Protocol Driver.
2. The Channel Manager Protocol Driver checks the channel map information it has obtained from the Service Information Tables. If the requested channel occurs on a frequency other than the one which is currently tuned, the Channel Manager Protocol Driver requests the Tuner Device Driver to tune to the appropriate frequency. This request to the Tuner Device Driver is specified as an asynchronous request with notification via a call-back function specified by the Channel Manager Protocol Driver.

If the request to change channels issued in **1** was specified to execute asynchronously, the Channel Manager Protocol Driver returns control to the calling application after the tune request has been issued. If the original request was specified to execute synchronously, the Channel Manager Protocol Driver suspends execution of the calling application, allowing another process to gain control of the CPU.

3. If the tuner hardware generates an interrupt when the tune operation has completed, the Tuner Device Driver's interrupt service routine notifies the Channel Manager Protocol Driver that the tune request has finished by invoking the call-back function specified by the Channel Manager Protocol Driver in **2**.

   If the tuner hardware is able to perform a tune operation immediately, the call-back to the Channel Manager Protocol Driver occurs during the execution of **2**.

4. The Channel Manager's tuner call-back function requests the Real-Time Device Driver to begin play-out of the selected program. This request to the Real-Time Device Driver is specified as an asynchronous request with notification via a call-back function specified by the Channel Manager Protocol Driver.

   In response, the Real-Time Device Driver registers a request for the Program Association Table and returns control to the Channel Manager Protocol Driver's tuner call-back function, and this function exits.

   When the Program Association Table has been received, the Real-Time Device Driver parses the table to obtain the Program Map Table's packet ID for the selected program and registers a request for this Program Map Table.

5. When the Program Map Table has been received, if there is a Condtional Access Driver in the system, the Real-Time Driver issues a request to the Conditional Access Device Driver to authorize and decrypt the selected channel. This request is specified as an asynchronous request with notification via invocation of a call-back function specified by the Real-Time Driver. The PMT Table is passed to the Conditional Access driver as part of the request.

6. In response to the authorization and decryption request issued in **5**, the Conditional Access Device Driver may need to obtain information from tables in the MPEG-2 transport stream. Typically only the PMT and the CAT tables are required. In any case, the Conditional Access Device Driver requests the PSI information it needs from the Real-Time Device Driver. The Real-Time Device Driver registers the request and returns control to the Conditional Access Device Driver.
7. When the information required by the Conditional Access Device Driver has been obtained from the network, the Real-Time Device Driver passes the information to the Conditional Access Device Driver's `dr_updata` function. This call executes in the context of the SPF's receive thread process.

   The Conditional Access Device Driver passes the necessary information from the MPEG-2 bitstream to the conditional access hardware and requests the hardware to decrypt the selected program.
8. If the conditional access hardware authorizes reception of the selected program it begins decryption and notifies the Real-Time Driver In response, the Conditional Access Device Driver invokes the Real-Time Driver's call-back function specified in **5**.

   For some conditional access hardware, the notification that authorization has been granted and the program is being decrypted is via an interrupt. In this case, **8** occurs in the Conditional Access Device Driver's interrupt service routine. For other conditional access hardware, the notification is immediate. In this case, **8** occurs during the call to the Conditional Access Device Driver's `dr_updata` function invoked in **7**.
9. When the authorization has been received from the Conditional Access Driver, the Real-Time Driver invokes the call-back function specified by the Channel Manager in **4**. If no Conditional Access System is present (e.g. in the Digital Terrestrial environment), then the steps 5-8 are skipped.
10. The channel change operation is now complete. If the original channel change request issued in **1** was specified to execute synchronously, the Channel Manager Protocol Driver's call-back function wakes-up the

suspended application. If the original request was specified to execute asynchronously, the Channel Manager Protocol Driver's callback function issues the appropriate notification to the application.

Chapter 6: The Channel Manager Protocol Driver

This chapter describes how to configure the device descriptor for the DVB and ATSC Channel Manager Protocol Driver provided with the Digital Broadcast Environment Pak.

RadiSys.
MICROWARE SOFTWARE

Configuring the Channel Manager Protocol Driver

The Channel Manager Protocol Driver has several configurable parameters defined in the driver's `defs.h` file. These parameters are described in this section.

The Channel Manager Driver Debug Module Name

The name of the debug module can be configured through the `CH_DEBUG_NAME` macro.

The Real-Time Driver's Device Descriptor Name

The name of the Real-Time Driver's device descriptor can be configured through the `RT_NAME` macro.

The Tuner Driver's Device Descriptor Name

The name of the Tuner Driver's device descriptor can be configured through the `TUNER_NAME` macro.

The Channel Map File Name

The name that the Channel Manager uses to save and restore the channel map to and from the NRF non-volatile RAM file system can be configured through the `CHAN_MAP_FILE_NAME` macro.

The Configuration File Name

The name that the Channel Manager uses to save and restore the configuration data and user preference settings to and from the NRF non-volatile RAM file system can be configured through the `CONFIG_FILE_NAME` macro.

The DBE User Preferences

The initial value of the user preference settings can be set through the `PREF_DESC macro`.

For More Information

The value of the `PREF_DESC` macro is a `ch_user_pref` structure.

The Maximum Number of Favorite Channel Rings

The maximum number of favorite channel rings can be configured through the `MAX_NBR_FAV_RINGS` macro.

The Maximum Number of Frequencies

The maximum number of frequencies which can be received can be configured through the `MAX_NBR_FREQUENCIES` macro.

The Maximum Number of Channels

The maximum number of channels which can be received can be configured through the `MAX_NBR_CHANS` macro.

The Maximum Number of Channels per Favorite Channel Ring

The maximum number of channels per favorite channel ring can be configured through the `MAX_NBR_CHAN_PER_FAV_RING` macro.

The Non-Volatile RAM usage flag

By changing the macro `USE_NV_RAM`, the Channel Manager driver can be configured to use or not use NVRAM storage to store the channel map. This setting can be overriden during the `nav_init()` call.

The Cache Size for the Current Event

The amount of memory allocated for the cache containing the current event can be changed using the macro `CURRENT_EVENT_CACHE_SIZE`. This cache contains current events on all channels and therefore is recommended to be at least 4k.

Chapter 7: The Tuner Device Driver

This chapter explains how to use the Tuner Device Driver provided with the Digital Broadcast Environment Pak to write a driver compatible with your tuner hardware.

Tuner Device Driver Basics

Purpose

The Tuner Device Driver in the DBE Pak architecture is responsible for changing the tuned frequency during channel change operations.

Since the Tuner Device Driver is a hardware driver, it must be ported to the actual tuner hardware present in your system. The Tuner Device Driver provided with the DBE Pak does not support any specific tuner hardware, but serves as a template driver which can be used as a starting point for writing a real tuner device driver. The template Tuner Device Driver is provided in source-code form.

Architecture

The Tuner Device Driver is written as a hardware SPF network device driver. However, unlike most SPF drivers, the Tuner Device Driver does not support any transfer of data. The sole purpose of the Tuner Device Driver is to tune to a specified frequency during channel change operations. Therefore, the SPF `dr_updata()` and SPF `dr_downdata()` entry points in the template Tuner Device Driver return an `EOS_UNKSVC` error code.

Tuning operations are performed through an SPF setstat request to the driver. For some tuner hardware, the tuning operation may complete immediately, whereas other tuner hardware may respond with an interrupt when the tune operation has completed. The template Tuner Device Driver assumes the tuner hardware generates an interrupt when a tuning request has completed. However, this driver can also be easily ported to hardware which responds to a tuning request immediately.

Requests to tune to a specific frequency may be specified as either synchronous or asynchronous. Asynchronous requests return to the caller immediately. For these requests, the caller uses the `notify_type` structure to specify the type of notification it wishes to receive when the tune operation has completed, and passes the address of this structure to the Tuner Device Driver during the channel change operation.

Using the Tuner Device Driver

Processing a Tuning Request

The Tuner Device Driver should first register the notification request by calling the `register_notify()` function. It should then program the hardware to perform the requested tuning operation. Finally, when tuning has completed, the tuner driver should call the `send_notify()` function to notify the caller that the tune request has finished.

For tuner hardware that responds immediately, the `send_notify()` function can be called within the `dr_setstat()` function after the hardware has been programmed. For tuner hardware indicating the completion of the tune request via a hardware interrupt, the driver's interrupt service routine should call the `send_notify()` function. The template Tuner Device Driver contains example code for the latter case.

For More Information

For information on using the `notify_type` structure, see *Chapter 5: The DBE Pak Device Driver Architecture*.

Structures

Two structures are used by the Tuner Device Driver. They are:

- `ite_ch_pb`
- `tuner_pb`

ite_ch_pb

Declaration

The `ite_ch_pb` structure is declared in the file `SPF/ch_mgr.h` as follows:

```
typedef struct ite_ch_pb
{
    spf_ss_pb   spb;
    u_char      flag1;
    u_char      flag2;
    u_int16     rsvd;
    Notify_type npb;
    void        *param1;
    void        *param2;
    void                 *param3;
    void                 *param4;
} ite_ch_pb, *Ite_ch_pb;
```

Description

The `ite_ch_pb` structure is used to pass parameters for various setstat requests in the DBE Pak. The Tuner Device Driver uses this structure for the `ITE_CH_TUNE` setstat request.

Fields

The fields in the `ite_ch_pb` structure are defined as follows:

`spb` is the standard SPF setstat parameter block structure. Within this structure:

- `code` specifies the requested setstat operation.
- `param` is a pointer to request-specific data. The use of the `param` field is described on a case-by-case basis in the specific setstat descriptions.

`size` is set to the size of data to which the `param` field points.

`updir` is always set to:

`0` when the setstat request is being passed from an application or higher-layer driver down the SPF protocol stack.

`1` when the setstat request is being passed up the protocol stack.

`flag1` is currently not used by the Tuner Device Driver.

`flag2` is currently not used by the Tuner Device Driver.

`rsvd` is reserved for future use. Entities using the Tuner Device Driver should set this field to `0` to remain compatible with future versions of the driver.

`npb` is a pointer to a `notify_type` structure owned by the caller.

For More Information

The `notify_type` structure is described in *Chapter 5: The DBE Pak Device Driver Architecture*.

`param1` is currently not used by the Tuner Device Driver.

`param2` is currently not used by the Tuner Device Driver.

`param3` is currently not used by the Tuner Device Driver.

`param4`	is currently not used by the Tuner Device Driver.

Declaration

The `tuner_pb` structure is declared in the file `SPF/tuner.h` as follows:

```
struct tuner_pb
{
   u_int32                  struct_type;
   u_char                   tuner_type;
   u_char                   tuner_index;
   u_char                   rsvd[2];

   union
   {
                            struct
                            {
                            u_charfrequency[4];
                            u_charflags[3];
                            u_charsymbol_rate[3];
                            u_charsymbol_fec_inner;
                            u_charreserved[5];
                            } dvb;

                            struct
                            {
                            u_int32freq;
                            booleandigital;
                            booleanaccess_controlled;
                            u_charmodulation_mode;
                            u_charuser_def[2];
                            u_charrsvd;
                            } atsc_psip;
   } tune;
};
```

Description

The tuner_pb structure is used to pass tuning parameters to the Tuner Device Driver's ITE_CH_TUNE setstat request.

Fields

The fields in the tuner_pb structure are defined as follows:

struct_type
: defines whether the tune union carries tuning information for the DVB digital broadcast standard or for the ATSC digital broadcast standard. This field should be set to either TUNER_STRUCT_DVB for DVB environments or TUNER_STRUCT_ATSC for ATSC environments.

tuner_type
: defines whether the tuner device being requested is for a Terrestrial, Satellite or Cable environment. This field can be set to either TUNER_TYPE_TERRESTRIAL, TUNER_TYPE_CABLE or TUNER_TYPE_SATELLITE.

tuner_index
: This field is used to specify the index of a tuner device, if multiple tuners of the same tuner_type are handled by the same driver. This field is set to a value from 0 to number of tuner devices of a particular type minus one.

dvb
: **For DVB Environments:** the dvb member structure of the tune union should be filled in with the DVB tuning information. This structure is 16 bytes long. Obtain these bytes as follows:

- For cable environments, copy the first 11 bytes directly from the last 11 bytes of cable delivery system descriptor in the Network Information Table.

- For satellite environments, copy the first 11 bytes directly from the last 11 bytes of the satellite delivery system descriptor in the Network Information Table.

Note

Macros in the file `SPF/tuner.h` can be used by the Tuner Device Driver to extract the individual fields from these 11 bytes.

`atsc`

For ATSC Environments: the `atsc` member structure of the `tune` union should be filled in with the ATSC tuning information. For analog frequencies the `frequency` field should be 1.25 MHz above the lower edge of the frequency's RF band. For digital frequencies, the `frequency` field should be 310 KHz above the lower edge of the frequency's RF band. The `access_controlled` field can be ignored.

Setstat Calls Supported

Most of the functionality provided by the Tuner Device Driver is implemented through the driver's `dr_setstat()` function. A pointer to an `spf_ss_pb` structure is passed as a parameter to this function. The `code` field in this structure specifies the requested operation. The different operations can be grouped into the following two categories:

- Standard SPF Setstat Calls
- Tuner Device Driver Specific Setstat Calls

Standard SPF Setstat Calls

The Tuner Device Driver supports the following standard SPF setstat functions:

Table 7-1 Standard SPF Setstat Calls

Call	Description
`SPF_SS_CLOSE`	Close a path to the Channel Manager Protocol Driver.
`SPF_SS_OPEN`	Open a path to the Channel Manager Protocol Driver.
`SPF_SS_POP`	Pop a driver from the protocol stack.
`SPF_SS_PUSH`	Push a driver onto the protocol stack.

These setstat functions should only be issued by the SPF File Manager, and therefore are not documented further in this manual.

For More Information

For information on these calls, refer to the ***SPF Porting Guide***.

Tuner Device Driver Specific Setstat Calls

The second category of setstat functions is for tuning to a specified frequency. The call `ITE_CH_TUNE` is the only setstat function included in this category.

ITE_CH_TUNE

Tune to New Frequency

Description

The ITE_CH_TUNE setstat requests the Tuner Device Driver to tune to a new frequency.

Parameters

For this setstat function, the caller passes a pointer to an ite_ch_pb structure to the Tuner Device Driver. The parameters for the ite_ch_pb structure are as follows:

spb	Within the spb structure: code must be set to ITE_CH_TUNE. param must point to the caller's tuner_pb structure. The tuner_pb structure defines the tuning parameters for the tune request. The remaining fields of the spb structure are not used for the ITE_CH_TUNE setstat request.
flag1	is not used for the ITE_CH_TUNE setstat.
flag2	is not used for the ITE_CH_TUNE setstat.
rsvd	is reserved for future use. Fill this field in as 0 to remain compatible with future versions of the driver.
npb	is a pointer to a notify_type structure owned by the caller. For the ITE_CH_TUNE setstat function, this structure specifies whether the call should execute synchronously or asynchronously. For the asynchronous case this structure also specifies the notification mechanism desired by the caller.

For More Information

The `notify_type` structure is described in *Chapter 5: The DBE Pak Device Driver Architecture*.

`param1`	is not used for the `ITE_CH_TUNE` setstat.
`param2`	is not used for the `ITE_CH_TUNE` setstat.
`param3`	is not used for the `ITE_CH_TUNE` setstat.
`param4`	is not used for the `ITE_CH_TUNE` setstat.

Getstat Calls Supported

Standard SPF Getstat Calls

The Tuner Device Driver supports the `SPF_GS_UPDATE` standard SPF getstat function. `SPF_GS_UPDATE` should only be issued by the SPF File Manager, and therefore is not documented further in this manual.

For More Information

For more information on the `SPF_GS_UPDATE` call, refer to the ***SPF Porting Guide***.

The Tuner Device Driver also supports the `TUNER_GS_GET_SIG_INFO` getstat request to retrieve information about the currently tuned frequency, and the `TUNER_GS_GET_CONFIGURATION` getstat to get the tuner configuration. For more information on `TUNER_GS_GET_SIG_INFO` see the `nav_get_signal_info()` API call in Chapter 3. The `TUNER_GS_GET_CONFIGURATION` getstat is described below.

TUNER_GS_GET_CONFIGURATION

Get Tuner Driver Configuration

Description

This getstat requests the Tuner Device Driver to return its configuration. This configuration is specified in the `tuner_config` structure specified below. The structure contains the number of terrestrial, cable and satellite tuner devices present in the system. This information is obtained and used by the Channel Manager and allows the same Channel Manager to be used in a Satellite, Cable or Terrestrial environment.

tuner_config

```
struct _tuner_config
{
    u_int32     nbr_terrestrial_inputs;
    u_int32     nbr_cable_inputs;
    u_int32     nbr_satellite_inputs;
};
```

TUNER_GS_GET_SIG_INFO

Get Signal Strength

Description

This getstat requests the Tuner Device Driver to return the signal strength on the current frequency. The value returned by the tuner driver is driver dependent and not specified by the DBE Pak. This getsta is used by applications performing the initial installation and setup of the SetTop Box. In a terrestrial environment, the application may need to do a frequency scan of all frequencies in the digital spectrum. Then based on the values returned by this getstat, it can filter out invalid frequencies. In Satellite and Cable environments, this getstat can be used to implement a meter based strength reading application that can be used by installation technicians to for e.g. adjust the satellite dish position.

Configuring the Tuner Device Driver

The template Tuner Device Driver has several parameters configured in the driver's `defs.h` file. These parameters are described in this section.

The Debug Module

When compiled for debug mode, the Tuner Device Driver writes debug text strings to a data module in memory. This module can then be examined to help determine the flow of control through the driver. The name of the data module used for debugging is set in the `lu_dbg_name` field of the driver's logical unit static storage memory.

Note

A debug version of the Tuner Device Driver can be created by using the `spfdbg.mak` makefile to compile the driver.

The Interrupt Vector

For tuner hardware using an interrupt to signal the completion of a tune request, the hardware's interrupt vector can be set via the `lu_vector` field in the driver's logical unit static storage. For hardware that is not interrupt driven, this field should be removed.

The Interrupt Priority Level

For tuner hardware using an interrupt to signal the completion of a tune request, the hardware's interrupt priority can be set via the `lu_priority` field in the driver's logical unit static storage. For hardware that is not interrupt driven, this field should be removed.

The Separate Tuners Flag

If the Analog and Digital Tuners are separate and can be used at the same time, i.e. the receiver can receive a digital bit-stream while being tuned to an analog channel, then the `SEPARATE_TUNERS` macro should be set to true.

The Tuner Configuration

The default tuner configuration can be changed using the `lu_tuner_config` field.

Skeleton Driver Fields

The `lu_tpb` and `lu_isr_delay_in_msec` fields in the driver's logical unit static storage are used to emulate an interrupt for the template skeleton device driver. These fields should be removed in a real tuner driver.

Chapter 8: The Conditional Access Device Driver

This chapter explains how to use the Conditional Access Device Driver provided with the Digital Broadcast Environment Pak to write a driver compatible with your decryption hardware. If there is no decryption hardware on your system then this driver is not required and can be left out without affecting the rest of the system.

Conditional Access Device Driver Basics

Purpose

In the DBE Pak architecture, the Conditional Access Device Driver is responsible for communicating with a system's conditional access hardware. In a typical system, the conditional access hardware includes a dedicated microprocessor. This processor is responsible for performing the actual authorization and decryption functions for the selected MPEG program. The conditional access processor performs these functions by obtaining and parsing certain information from tables in the incoming MPEG-2 transport stream. Typically, it is the responsibility of the Conditional Access Device Driver to request these tables from the Real-Time Device Driver and pass the necessary information from the tables to the conditional access hardware's processor.

The Conditional Access Device Driver is a hardware driver and must be ported to the actual conditional access hardware present in your system. The Conditional Access Device Driver provided with the DBE Pak does not support any specific conditional access hardware. Instead, this driver serves as a template driver to be used as a starting point for writing a real conditional access device driver. The template Conditional Access Driver is provided in source-code form.

Architecture

The Conditional Access Device Driver is written as an SPF hardware device driver. However, unlike most SPF hardware drivers, the Conditional Access Device Driver does not transfer data to or from a network interface. The Conditional Access Device Driver may, however, relay certain information from the appropriate MPEG-2 EMM stream and Program Map Table to the conditional access hardware. If this is required, the Conditional Access Device Driver opens a dedicated path to the Real-Time Device Driver to request the necessary information from the incoming MPEG-2 transport stream. Entities using the Conditional Access Device Driver are unaware of the existence of this path. The Real-Time Device Driver passes the requested tables to the Conditional Access Device Driver, a section at a time, in mbufs via the Conditional Access Device Driver's `dr_updata()` function.

Similarly, in the DBE Pak architecture the Channel Manager Protocol Driver opens a dedicated path to the Conditional Access Device Driver. Applications using the Channel Manager Protocol Driver are unaware of the existence of this path. The Channel Manager Protocol Driver uses this path to issue requests to start or stop decryption of a program to the Conditional Access Device Driver. These requests are issued solely via the Conditional Access Device Driver's `dr_setstat()` entry point. No data is passed from the Channel Manager Protocol Driver to the Conditional Access Device Driver via the Conditional Access Device Driver's `dr_downdata()` entry point. Similarly, no data is passed from the Conditional Access Device Driver to the Channel Manager Protocol Driver via the Channel Manager Protocol Driver's `dr_updata()` entry point.

Although the flow of information depends on the requirements of the actual conditional access hardware, a typical conditional access driver performs the following functions:

- Upon initialization, the Conditional Access Device Driver opens a path to the Real-Time Device Driver and issues a request to the Real-Time Device Driver to obtain the Conditional Access Table. The Conditional Access Table is always carried in MPEG-2 transport packets with packet IDs of one. The Conditional Access Device Driver uses the Conditional Access Table to locate the EMM stream packet ID.
- When the Conditional Access Table is received, the Conditional Access Device Driver parses this table and then issues a request to the Real-Time Device Driver to obtain the EMM stream. As EMM messages arrive, the Conditional Access Device Driver passes them to the conditional access hardware. These EMM messages contain information the conditional access hardware uses to authorize or deny decryption of specific programs for individual decoders.
- When a channel change request occurs, the Channel Manager Protocol Driver issues a setstat call to the Conditional Access Device Driver to abort decryption of the previous channel, followed by a setstat call to request decryption of the new channel.
- When requested to begin decryption for a new channel, the Conditional

Access Device Driver requests the conditional access descriptors for the new program from the Real-Time Device Driver. This request may be made through an `ITE_GET_PGMINFO` setstat call to the Real-Time Device Driver or by requesting the PMT from the Real-Time driver.

- When the program information is received, the Conditional Access Device Driver extracts the conditional access descriptors and passes them to the conditional access hardware. If the conditional access hardware determines the decoder is authorized to receive the new program, it uses these Conditional Access Descriptors to locate the ECM streams' packet IDs for the selected program. The ECM streams contain decryption keys for the elementary streams comprising the selected program.
- Once the ECM streams' packet IDs are determined, typical conditional access hardware automatically extracts the ECM packets from the incoming bit stream, obtains the decryption keys, and performs decryption of the elementary streams belonging to the selected service.

Requests from the Channel Manager Protocol Driver to begin decryption of a new service can be specified as either synchronous or asynchronous. For asynchronous requests, the Conditional Access Device Driver returns control to the Channel Manager Protocol Driver immediately. The Channel Manager Protocol Driver specifies the request type (synchronous or asynchronous) as well as the notification method for asynchronous requests by filling in a `notify_type` structure and passing the address of this structure to the Conditional Access Device Driver as part of the setstat request for decryption.

For More Information

For information on using the `notify_type` structure, see *Chapter 5: The DBE Pak Device Driver Architecture*.

Structures

The `ite_ch_pb` structure is supported by the Conditional Access Device Driver.

Declaration

The `ite_ch_pb` structure is declared in the file `SPF/ch_mgr.h` as follows:

```
typedef struct ite_ch_pb
{
    spf_ss_pb           spb;
    u_char              flag1;
    u_char              flag2;
    u_int16             rsvd;
    Notify_type         npb;
    void                *param1;
    void                *param2;
} ite_ch_pb, *Ite_ch_pb;
```

Description

The `ite_ch_pb` structure is used to pass parameters for various setstat requests in the DBE Pak. The Conditional Access Driver uses this structure for the `ITE_CH_CA_ABORT` and `ITE_CH_CA_START` setstat requests.

Fields

The fields in the `ite_ch_pb` structure are defined as follows:

`spb` is the standard SPF setstat parameter block structure. Within this structure:

- `code` specifies the requested setstat operation.
- `param` is a pointer to request-specific data. The use of the `param` field is described on a case-by-case basis in the specific setstat descriptions.
- `size` is set to the size of data to which the `param` field points.

`updir` is always set to:

- `0` when the setstat request is being passed from an application or higher-layer driver down the SPF protocol stack.
- `1` when the setstat request is being passed up the protocol stack.

`flag1` is currently not used by the Conditional Access Device Driver.

`flag2` is currently not used by the Conditional Access Device Driver.

`rsvd` is reserved for future use. Entities using the Conditional Access Device Driver should fill this field in as `0` to remain compatible with future versions of the driver.

`npb` is a pointer to a `notify_type` structure owned by the caller.

For More Information

The `notify_type` structure is described in *Chapter 5: The DBE Pak Device Driver Architecture*.

`param1` is currently not used by the Conditional Access Device Driver.

`param2` is currently not used by the Conditional Access Device Driver.

Setstat Calls Supported

Most of the functionality provided by the Conditional Access Device Driver is accessed through the driver's `dr_setstat()` entry point. A pointer to an `spf_ss_pb` structure is passed as a parameter to this function. The `code` field in this structure specifies the requested operation. The different operations can be grouped into the following two categories:

- Standard SPF Setstat Calls
- Conditional Access Device Driver Specific Setstat Calls

Standard SPF Setstat Calls

The Conditional Access Device Driver supports the following standard SPF setstat requests:

Table 8-1 Standard SPF Setstat Calls

Call	Description
`SPF_SS_CLOSE`	Close a path to the Conditional Access Device Driver.
`SPF_SS_OPEN`	Open a path to the Conditional Access Device Driver.
`SPF_SS_POP`	Pop a driver from the protocol stack.
`SPF_SS_PUSH`	Push a driver onto the protocol stack.

These setstat requests should only be issued by the SPF File Manager, and therefore are not documented further in this manual.

For More Information

For information on these calls, refer to the ***SPF Porting Guide***.

Conditional Access Device Driver Specific Setstat Calls

The Conditional Access Device Driver supports the following setstat requests for controlling decryption of a specified service:

Table 8-2 Decryption Setstat Calls

Call	Description
`ITE_CH_CA_ABORT`	Stop decryption of current service.
`ITE_CH_CA_START`	Begin decryption of a new service.

Description

The ITE_CH_CA_ABORT setstat requests the Conditional Access Device Driver to stop decryption of the current service.

Parameters

The caller must pass a pointer to an ite_ch_pb structure to the Conditional Access Device Driver rather than a pointer to an spf_ss_pb structure. However, the first parameter of the ite_ch_pb structure is the standard spf_ss_pb SPF setstat structure. The parameters for the ite_ch_pb structure are as follows:

spb	Within the spb structure: code must be set to ITE_CH_CA_ABORT. param must be filled in with the service number (program number) of the program for which decryption is to stop. The remaining fields of the spb structure are not used for the ITE_CH_CA_ABORT setstat request.
flag1	is not used for the ITE_CH_CA_ABORT setstat.
flag2	is not used for the ITE_CH_CA_ABORT setstat.
rsvd	is reserved for future use. Fill this field in as 0 to remain compatible with future versions of the driver.
npb	is not used for the ITE_CH_CA_ABORT setstat.
param1	is not used for the ITE_CH_CA_ABORT setstat.

`param2`	is not used for the `ITE_CH_CA_ABORT` setstat.

SEE ALSO

`ITE_CH_CA_START`

Description

The ITE_CH_CA_START setstat requests the Conditional Access Device Driver to authorize and start decryption of a new service.

Parameters

The caller must pass a pointer to an ite_ch_pb structure to the Conditional Access Device Driver rather than a pointer to an spf_ss_pb structure. However, the first parameter of the ite_ch_pb structure is the standard spf_ss_pb SPF setstat structure. The parameters for the ite_ch_pb structure are as follows:

spb	Within the spb structure: code must be set to ITE_CH_CA_START. param must be filled in with the service number (program number) of the program for which decryption is to start. The remaining fields of the spb structure are not used for the ITE_CH_CA_START setstat request.
flag1	is not used for the ITE_CH_CA_START setstat.
flag2	is not used for the ITE_CH_CA_START setstat.
rsvd	is reserved for future use. Fill this field in as 0 to remain compatible with future versions of the driver.
npb	is a pointer to a notify_type structure owned by the caller. This structure specifies whether the call should execute synchronously or asynchronously. For the

asynchronous case this structure also specifies the notification mechanism desired by the caller.

For More Information

The `notify_type` structure is described in *Chapter 5: The DBE Pak Device Driver Architecture*.

`param1`	is not used for the `ITE_CH_CA_START` setstat.
`param2`	is not used for the `ITE_CH_CA_START` setstat.

SEE ALSO

`ITE_CH_CA_ABORT`

Getstat Calls Supported

Standard SPF Getstat Calls

The Conditional Access Device Driver supports the `SPF_GS_UPDATE` standard SPF getstat function. This getstat function should only be issued by the SPF File Manager, and therefore is not documented further in this manual.

For More Information

For information on the `SPF_GS_UPDATE` call, refer to the ***SPF Porting Guide***.

Configuring the Conditional Access Device Driver

The template Conditional Access Device Driver has several parameters configured in the driver's `defs.h` file. These parameters are described in this section.

The Real-Time Device Driver's Device Descriptor Name

For conditional access drivers which must open a path to the Real-Time Device Driver, the name of the Real-Time Device Driver's device descriptor is defined with the `RT_NAME` macro.

The Debug Module

When compiled for debug mode, the Conditional Access Device Driver writes debug text strings to a data module in memory. This module can then be examined to help determine the flow of control through the driver. The name of the data module used for debugging is set in the `lu_dbg_name` field of the driver's logical unit static storage memory.

Note

A debug version of the Conditional Access Device Driver can be created by using the `spfdbg.mak` makefile to compile the driver.

The Interrupt Vector

For conditional access hardware using interrupts, the hardware's interrupt vector can be set via the `lu_vector` field in the driver's logical unit static storage. For hardware that is not interrupt-driven, this field should be removed.

The Interrupt Priority Level

For conditional access hardware using interrupts, the hardware's interrupt priority can be set via the `lu_priority` field in the driver's logical unit static storage. For hardware that is not interrupt-driven, this field should be removed.

Skeleton Driver Fields

The `lu_tpb` and `lu_hw_delay_in_msec` fields in the driver's logical unit static storage are used to emulate an interrupt for the template conditional access device driver. These fields can be removed in a real conditional access driver.

Chapter 9: The Real-Time Network Driver

This chapter explains how to configure the Real-Time Network Driver provided with the Digital Broadcast Environment Pak.

Real-Time Network Driver Basics

Purpose

The DBE Pak contains an updated version of the DAVID and DAVID*Lite* Real-Time Network Driver. This driver provides several different functions to other device drivers or application programs in the DBE Pak architecture including:

- A set of calls allowing other drivers or application programs to obtain SI, PSI, and private data from an incoming MPEG-2 transport stream.
- A set of calls allowing other drivers or application programs to start and stop the decoding and playout of an MPEG program. The desired program can be specified by either passing the Real-Time Network Driver the list of Packet Identifiers (PIDs) for the elementary streams to be decoded, or by passing the program number of the desired program. In the latter case, the Real-Time Network Driver obtains and parses the Program Association Table and Program Map Table to determine the elementary stream PIDs to be decoded.
- A set of calls for reading and setting the current language preferences. These preferences are used by the Real-Time Network Driver to select a language track on a multi-language program when the Real-Time Network Driver chooses the audio elementary stream to be decoded for a program.
- A set of calls allowing other drivers or application programs to obtain detailed information about a specific MPEG program. For these calls the Real-Time Network Driver obtains and parses the Program Association Table and Program Map Table from the network and initializes a data structure provided by the requesting entity with information about the requested program.
- A set of calls allowing applications to read data from an incoming MPEG-2 transport stream using Stream Control Blocks and Stream Control Lists. These calls provide backward compatibility with previous versions of the Real-Time Network Driver.

Architecture

The Real-Time Network Driver is written as a standard SPF network device driver. In the DBE Pak architecture, the Channel Manager Protocol Driver, Conditional Access Device Driver, and EPG-Data API all make use of the functions provided by the Real-Time Network Driver.

When the Real-Time Network Driver is initialized, it establishes a link to the DUXMAN Device Manager. It then issues requests to DUXMAN to obtain specific SI, PSI, or private data from the incoming MPEG-2 transport stream. The requested data must conform to the MPEG-2 section syntax. DUXMAN filters the incoming bit stream to extract the requested data and passes this data to the Real-Time Network Driver in mbufs. Each mbuf normally contains a single, complete MPEG-2 section. However, when the Program Association Table is requested, DUXMAN returns all sections of the table concatenated together in a single mbuf.

Requests for MPEG-2 sections to DUXMAN generated by the Real-Time Network Driver may either be for certain PSI data required by the Real-Time Network Driver itself, or may be for data requested from the Real-Time Network Driver by higher-layer drivers in a protocol stack or by application programs. For data requested by higher-layer drivers or applications, the Real-Time Network Driver passes the mbufs received from DUXMAN to the appropriate entity. When the requesting entity is a higher-layer driver in a protocol stack, the data is passed to that driver's `dr_updata()` function. When the requesting entity is an application, the data is passed to the SPF File Manager. In this case, the application can read the data using the standard OS-9 or OS-9000 I/O system calls.

In the event more than one entity has issued requests to the Real-Time Network Driver for the same data, the Real-Time Network Driver issues a single request for this data to DUXMAN. When an mbuf containing data matching this request is received from DUXMAN, the Real-Time Network Driver generates a copy of this mbuf for each entity requesting the data and forwards these copied mbufs to the appropriate entities.

Configuring the Real-Time Network Driver

The Real-Time Network Driver has several parameters configured in the driver's `defs.h` file. These parameters are described in this section.

The DUXMAN Device Descriptor Name

The name of the DUXMAN Device Manager's device descriptor is defined in the `lu_duxman_name` field of the Real-Time Network Driver's logical unit static storage. This name must be less than or equal to 16 characters, including the NULL terminating byte.

The CA Device Descriptor Name

The name of the Conditional Access Driver's device descriptor is   defined by the macro `CA_NAME`. This name must be less than or equal to 16 characters, including the NULL terminating byte.

The Maximum Number of Stream Control Blocks

The maximum number of concurrent `ITE_RD_STREAM` requests is defined in the `lu_max_num_scbs` field of the Real-Time Network Driver's logical unit static storage.

The Default Language Preferences

The initial language preference settings is defined in the `lu_stream_pref` field of the Real-Time Network Driver's logical unit static storage. This field is an `ite_stream_pref` structure as defined in the file `mpeg2.h`.

The Debug Module

When compiled for debug mode, the Real-Time Network Driver writes debug text strings to a data module in memory. This module can be examined to help determine the flow of control through the driver. The name of the data module used for debugging is set in the `lu_dbg_name` field of the driver's logical unit static storage memory.

Note

A debug version of the Real-Time Network Driver can be created by using the `spfdbg.mak` makefile to compile the driver.

The Exclusive Access Flag

The Real-Time Network Driver can be configured to only accept `ITE_PLAY_PGM` and `ITE_ABT_PGM` setstat requests from the process and path used for the `ITE_PLAY_PGM` request which is currently active. Under this mode of operation, any `ITE_PLAY_PGM` or `ITE_ABT_PGM` setstat request from a process or path other than that used to set the currently playing program fails with an `E_DEVBSY` error code. If no program is currently playing, then any process/path can be used to issue an `ITE_PLAY_PGM` request.

This mode of operation is set through the `lu_decoder_exclusive_access` field in the driver's logical unit static storage memory. If this flag is set to a non-zero value, then the described mode is invoked. If this flag is zero, then any process or path can be used to issue the `ITE_PLAY_PGM` and `ITE_ABT_PGM` setstat requests.

Chapter 10: The Parental Control API

This chapter contains descriptions, in alphabetical order, of Parental Control functions.

Function Descriptions

The function descriptions are, for the most part, self-explanatory. Each function description contains the following sections:

The SYNTAX section shows the function prototype with the required parameters and their data types.

The DESCRIPTION section provides a description of the function.

The PARAMETERS section provides details about each of the function's parameters.

FATAL ERRORS are errors detected within the API call and are returned directly by that particular call. Applications may not be able to recover from fatal errors.

API specific Errors are errors detected within the API call and are a direct result of that particular call. Applications can recover from API specific Errors.

INDIRECT ERRORS are errors returned by another function called during the processing of the API request.

The SEE ALSO section lists related functions or materials that provide more information about the function.

Parental Control API Functions

The table below summarizes the Parental Control functions:

Table 10-1 Summary of Parental Control Functions

Function	Description
`pc_add_privilege()`	Add a privilege to an account.
`pc_create_account()`	Create a user account.
`pc_delete_account()`	Delete user account
`pc_delete_av_blackout_notification()`	Delete a notification set for A/V blackout
`pc_delete_privilege()`	Delete a privilege from an account.
`pc_disable()`	Disable Parental Control functionality.
`pc_enable()`	Enable Parental Control functionality.
`pc_get_account_info()`	Gets information from all the current user accounts.
`pc_init()`	Initialize the Parental Control API.
`pc_login()`	Attempt to login as a user.
`pc_logout()`	Log out the current user.

Table 10-1 Summary of Parental Control Functions (continued)

Function	Description
`pc_set_av_blackout_notification()`	Provides an Audio/Video blackout callback function.
`pc_set_password()`	Changes the password for a user account.
`pc_term()`	Terminates the Parental Control API.

pc_add_privilege()

Add a privilege to an account.

Syntax

```
#include <SPF/pc.h>

error_code pc_add_privilege
(
    char*                   user_login_name,
     pc_privilege*          privilege
);
```

Description

Add a privilege to an account. The user must be logged in as a parent account for this call to succeed.

Parameters

`user_login_name`	Login name of the user for whom the privileges are being set.
`privilege`	Pointer to the user privilege information. This structure is specified in pc.h

API specific Errors

`E_NOTRDY`	API not initialized.
`E_ILLARG`	Illegal or NULL Parameter

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`pc_delete_privilege()`

pc_create_account()

Create a user account

Syntax

```
#include <SPF/pc.h>

error_code pc_create_account
(
     char*          user_login_name,
     char*          password,
     boolean        is_parent,
     boolean        is_default_account,
     int            default_channel_privilege,
     int            default_time _privilege
);
```

Description

This call creates a user account specified by a login name and password. This call can only be made if the current user privilege has been set to a parent account using `pc_login()`

Parameters

`user_login_name`	Login name for the user account created.
`password`	Password for this account. This can be NULL for a child account.
`is_parent`	If set to TRUE, then the account is a parent account with unlimited viewing privileges.
`is_default_account`	If set to TRUE, then the account is the default or guest account which is used when a user login fails or is cancelled.

`default_channel_privilege`

If set to 0, then all channels are off limits unless specified in a privilege. If non-zero then every channel is accessible unless restricted in a privilege.

`default_time_privilege`

If set to 0, then the TV is off limits at all times of the day unless specified in a privilege. If set to non-zero, then the TV is always accessible except during times restricted by a privilege.

API specific Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_getstat()` and `_os_setstat()`.

See Also

`pc_delete_account()`

pc_delete_account()

Deletes the user account identified by user_login_name

Syntax

```
#include <SPF/pc.h>

error_code pc_delete_account
(
   char* user_login_name
);
```

Description

This call deletes the user account identified by `user_login_name`. The call only succeeds if the current user is a parent account. The current parent account that is logged in cannot be deleted.

Parameters

`user_login_name`	Login name for the user account to be deleted.

API specific Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`pc_create_account()`

pc_delete_av_blackout_notification()

Delete A/V blackout notification

Syntax

```
#include <SPF/pc.h>

error_code pc_delete_av_blackout_notification();
```

Description

This call deletes a requested A/V blackout notification. Note that the A/V blackout notification is persistent i.e it remains active once it is made until it is explicitly deleted using this call. Also only one notification can be active per application process, since the notification is managed on a per-path basis and the Parental Control API opens a path to the Channel Manager.

Parameters

`None`

API specific Errors

`EOS_NOTRDY`	API not initialized

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`pc_set_av_blackout_notification()`

pc_delete_privilege()

Delete a privilege from an account

Syntax

```
#include <SPF/pc.h>

error_code pc_delete_privilege
(
   char*user_login_name,
   pc_privilege*privilege

);
```

Description

Deletes a privilege from an account.

Parameters

`user_login_name`	Login name of the user for whom the privileges are being set.
`privilege`	Pointer to the user privilege information. This structure is defined in pc.h and must be allocated by the application.

API specific Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLARG`	One of the arguments passed was invalid.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`pc_add_privilege()`

pc_disable()

Disables Parental Control Functionality

Syntax

```
#include <SPF/pc.h>

error_code pc_disable()
```

Description

This call disables the parental control functionality. If parental control is disabled, then all user accounts remain intact, except that viewing is no longer regulated. When the control is re-enabled then the existing user accounts are revived and do not need to be created again. This is provided to give the application a low overhead mechanism for toggling parental control functionality.

Parameters

None

API specific Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLARG`	request_handle specified is `NULL` or invalid.

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`pc_enable()`

pc_enable()

Enables the Parental Control functionality

Syntax

```
#include <SPF/pc.h>

error_code pc_enable()
```

Description

This call enables the Parental Control functionality. If Parental Control is disabled, then all user accounts reamin intact, except that viewing is no longer regulated. When the control is re-enabled, then the existing user accounts are revives and do not need to be created again. This is provided to give the application a low overhead mechanism for toggling Parental Control functionality.

Parameters

None

API specific Errors

`EOS_NOTRDY`	API not initialized
`EOS_ILLARG`	Illegal parameter

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`pc_enable()`

pc_get_account_info()

Gets information on all the current user accounts

Syntax

```
#include <SPF/pc.h>

error_code pc_get_account_info
(
    pc_account**          ptr_account_list,
    pc_account**          ptr_current_account,
    pc_account**          ptr_default_account,
    boolean*              is_pc_enabled
);
```

Description

This call gets information on all the current user accounts. This information is stored in the driver. The call returns a pointer to the head os a link list maintained by the driver. Each entry in the link list is a user account structure. The application only has read permissions to the structures since this memory is owned by the driver.

This call also returns a pointer to the current account, the default account and indicates if parental control is enabled.

Parameters

`ptr_account_list`	Pointer to the head of the driver link list of user account structures. This is returned by the call
`ptr_current_account`	Pointer to the returned pointer to the current account
`ptr_default_account`	Pointer to the returned pointer to the default account
`is_pc_enabled`	Pointer to the returned flag that indicates if parental control is enabled

API specific Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLARG`	Illegal parameter passed in

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`pc_create_account()`

pc_init()

Initialize the Parental Control API

Syntax

```
#include <SPF/pc.h>
error_code pc_init
(
    char*           pc_param_file

);
```

Description

Initialize the Parental Control API. This call accepts the name of a configuration file, which contains the user account information. The name should include the complete path namefor e.g."/nv0/pc_params". If the file does not exist (this would be the case when the box is being programmed in the factory), then the file is created by the API. This call opens a path to the channel manger.

Parameters

`pc_param_file`	Name of configuration file containing user account information.

PI specific Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`pc_term()`

Syntax

```
#include <SPF/pc.h>
error_code pc_login
(
   char*                    user_login_name,
   char*                    password
);
```

Description

This call attempts to login as a user. The API makes no attempt to keep track of the number of times a login attempt is made. That responsibility is left to the application. If the login succeeds then this user becomes the current user and his/her privileges become the current privileges for the Set-top Box.

Parameters

`user_login_name`	login name for the user
`password`	Password for the user.

API specific Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLARG`	Illegal parameter passed in

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`pc_logout()`

pc_logout()

Logs out the current user

Syntax

```
#include <SPF/pc.h>
error_code pc_logout()
```

Description

This call logs out the current user. The system reverts to the default or guest use account if one exists, else no TV can be viewed unless another user logs in.

Parameters

None

API specific Errors

`EOS_NOTRDY`	API not initialized
`EOS_ILLARG`	Illegal parameter

Indirect Errors

Errors generated by `_os_setstat()`

See Also

`pc_login()`

Syntax

```
#include <SPF/pc.h>
error_code pc_set_av_blackout_notification
(
    Nav_notify      notify,
    Nav_status      status
);
```

Description

Since the Parental Control API has no control over the Set-top A/V hardware, the application can request a notification when a channel needs to be locked out for parental control reasons. When the Parental Control API detects that that the currently tuned to channel is out-of-bounds for the current user, it sends this notification. On receiving this notification, the application is responsible for blanking out the video signal, muting the audio hardware and displaying an appropriate OSD message.

Also when the Parental Control API detects that the channel can now be unlocked, it sends the same notification. The application can use the `status->err` field to detect if the notification was for blocking or unblocking the video. Under the current implementation, it is possible for the application to receive a unblocking notification on a channel that is already unblocked and vice versa. It is upto the application to keep track of the current state of the video hardware (blocked or unblocked) and ignore the notification if needed.

This notification is persistent, i.e the request for notification remains active until a `pc_delete_av_blackout()` notification call is made. Hence, this call needs to be made only once.

Parameters

`notify`	Pointer to the notification structure allocated and passed in by the application. This structure is defined in nav_api.h. It allows the user to specify which signal or event it wants.
`status`	Pointer to the status field of the notification. `status->err` is set to either `PC_SET_BLACKOUT` or `PC_REMOVE_BLACKOUT` depending on if the notification is for blocking or unblocking the video signal. The status field should be allocated by the application and needs to be kept allocated until the notification is deleted. Hence it is not advisable to allocate the status field on the stack.

API specific Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat().`

See Also

`pc_delete_av_blackout_notification()`

pc_set_password()

Changes the password for a user account

Syntax

```
#include <SPF/pc.h>

error_code pc_set_password
(
   char*user_login_name,
   char*old_password,
   char*new_password
);
```

Description

This call changes the password for a user account.

Parameters

`user_login_name`	login name of the user for whom the password informationis being changed
`old_password`	Old password for the account. If none exists, the NULL may be passed in.
`new_password`	New password for the account.

API specific Errors

`EOS_NOTRDY`	API not initialized.
`EOS_ILLARG`	Illegal parameter.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`pc_create_account()`

pc_term()

Terminates the Parental Control API

Syntax

```
#include <SPF/pc.h>
error_code pc_term()
```

Description

This call terminates the Parental Control API. The path to the configuration file and the path to the channel manager are closed.

Parameters

None.

API specific Errors

`EOS_NOTRDY`	API not initialized.

Indirect Errors

Errors generated by `_os_setstat()`.

See Also

`pc_init()`

Index

A

B

C

D

E

F

G

I

L

R

S

T

U

Product Discrepancy Report

To: Microware Customer Support

FAX: 515-224-1352

From:____

Company:_______________________________________

Phone:____

Fax:______________________Email:___________________

Product Name:

Description of Problem:

Host Platform______________________________________

Target Platform_____________________________________

RadiSys.

MICROWARE SOFTWARE