Using DUXMan

Version 2.2

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 C
July 1999

Copyright and publication information

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.

July 1999

Table of Contents

Chapter 1: Using DUXMan

This chapter provides an overview of DUXMan and includes the following sections:

- **Overview**
- **Special Features**
- **DUXMan System Architecture**
- **Using DUXMan**

Overview

The Demultiplexer Manager (DUXMan) is a device manager — a special device driver having no associated file manager. A device manager performs functions on behalf of other drivers in the system. DUXMan manages the Transport Demultiplexer Integrated Circuit (TDIC) in DAVID hardware devices.

DUXMan functions are called by other device drivers and file managers such as the Stacked Protocol File manager (SPF) and the Motion Picture File Manager (MPFM). DUXMan functions control:

- MPEG-2 transport stream demultiplexing
- MPEG stream system layer parsing
- Program Specific Information (PSI) extraction
- Delivery of audio/video/private data

DUXMan is not directly called by an application program.

Special Features

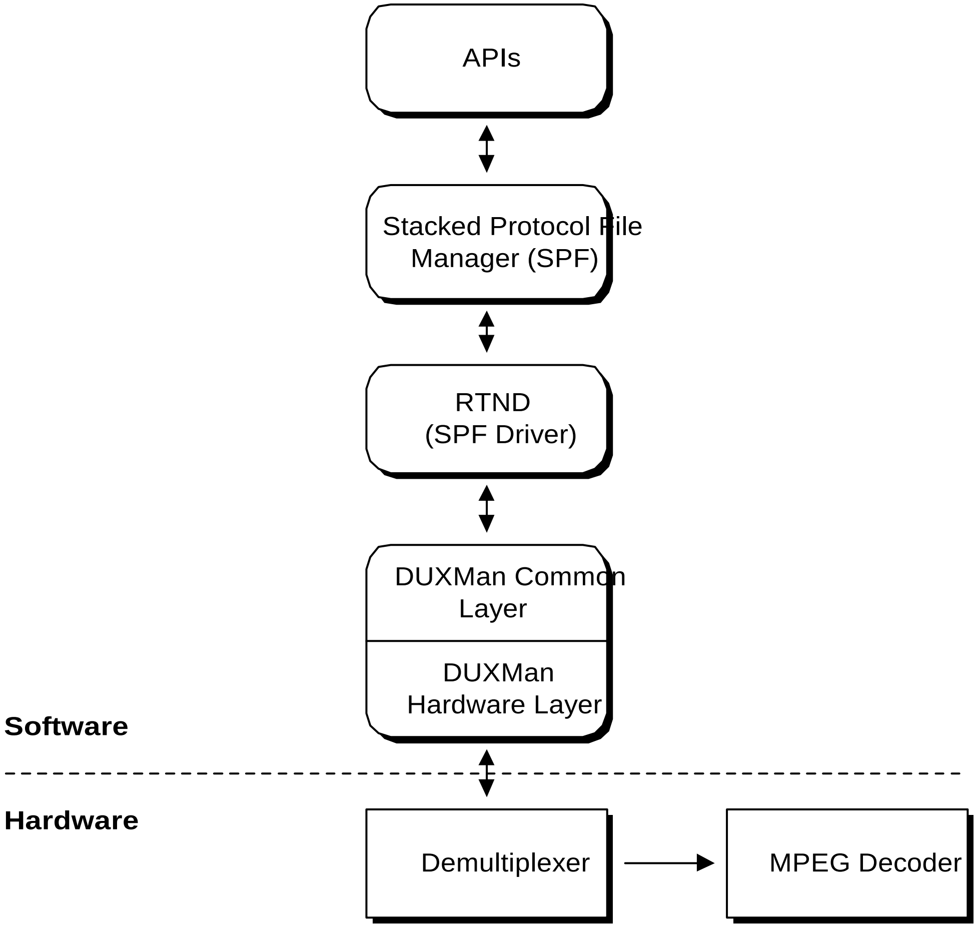
DUXMan has the following special features:

- The TDIC, under management of DUXMan, is not a standard I/O device. The TDIC is a bridge for other devices, such as the Network Interface Module (NIM) and the MPEG decoders. DUXMan functions are available to SPF, MPFM, and other device drivers.
- DUXMan provides faster communication between different driver subsystems because it does not have a file manager layer above it. DUXMan calls are entered using direct entry points into the system.
- DUXMan implements in software the functions not implemented by the TDIC if there is enough CPU bandwidth.
- DUXMan software contains two parts: the common layer and the port-specific or hardware layer. The common layer code is the same for all ports. For a specific port, only the port-specific layer must be customized.

DUXMan System Architecture

Figure 1-1 shows a block diagram of a DUXMan sub-system and its hardware environment.

Figure 1-1 Block Diagram of Major Components



Using DUXMan

To make DUXMan functionality available to a device driver or file manager, follow these steps:

Step 1. Include the following header files:

```
dvm.h
duxman.h
```

Step 2. Link to the following libraries:

```
os_dvm.l
os_dxm.l
```

Step 3. In the driver's initialization code, make the following call to pass the DUXMan device name and the address of the returned DUXMan device manager handle.

```
_os_dvm_link("dux", &dxmdata_pointer);
```

Step 4. After the initialization code has been executed, all DUXMan calls must pass the DUXMan device manager pointer as their first parameter.

Step 5. In the driver's de-initialization code, make the following call to unlink DUXMan.

```
_os_dvm_unlink("dux", dxmdata_pointer);
```

Chapter 2: DUXMan Programming Reference

This chapter presents the complete functional interface for DUXMan. It includes the following sections:

- **Introduction**
- **DUXMan Functional Interface**

Introduction

Synchronous and Asynchronous Calls

For a given Packet ID (PID), you may issue either multiple synchronous calls or multiple asynchronous calls, but not both simultaneously. If a synchronous call is active on a PID, no asynchronous call can be made on this same PID until the synchronous call is done, and vice versa.

Since DUXMan caches Program Association Tables (PAT), you can get the PAT through either the synchronous call `_os_dxm_pid_getpsi()` or the asynchronous call `_os_dxm_pid_getsect()`. The only difference is `_os_dxm_pid_getpsi()` gets whatever is available in the cache and `_os_dxm_pid_getsect()` waits until the requested version is available.

Callback Functions

Asynchronous functions pass data to the caller through callback functions. The `mbuf` mechanism passes the information to the caller. The callback function carries one complete section of data to the caller every time it is called for all tables with the exception of the PAT, where a complete table is passed to the caller.

DUXMan Functional Interface

The following table summarizes DUXMan functions for drivers and file managers that use DUXMan. Among them are two device manager calls labeled `_os_dvm_link()` and `_os_dvm_unlink()`, which initialize and deinitialize DUXMan, respectively. The rest of the calls are DUXMan-specific. After DUXMan is initialized, functions can be called directly. These functions can be called by any system-level software such as SPF or MPFM drivers.

Table 2-1 DUXMan Calls

Function	Description
`_os_dxm_pid_abtsect()`	Links to a Device Manager
`_os_dvm_unlink()`	Unlinks from a Device Manager
`_os_dxm_flush()`	Disables All Active PID Streams
`_os_dxm_flush_pat()`	Flushes PAT Cache
`_os_dxm_getstat()`	DUXMan Getstat Call
`_os_dxm_pid_abtsect()`	Abort GetSect Call
`_os_dxm_pid_addbuf()`	Adds Buffer for Private Data Output
`_os_dxm_pid_chgsect()`	Change Asynchronous GetSect Call
`_os_dxm_pid_delete()`	Deletes an Active PID Stream
`_os_dxm_pid_event()`	Registers or Removes an Event
`_os_dxm_pid_getpsi()`	Get the PSI Tables

Table 2-1 DUXMan Calls (continued)

Function	Description
`_os_dxm_pid_getsect()`	Get Section Data Asynchronously
`_os_dxm_pid_insert()`	Selects a Specific PID Stream to Output
`_os_dxm_pid_status()`	Gets the Status of a PID stream
`_os_dxm_setstat()`	DUXMan Setstat Call

_os_dvm_link()

Links to a Device Manager

Syntax

```
#include <dvm.h>
error_code _os_dvm_link(
     char         dev_name,
     void         **dvmp);
```

Libraries

```
os_dvm.l
```

Description

`_os_dvm_link()` links to and initializes a specific device manager such as DUXMan. This function returns a handle that in most cases is a pointer to the global storage area of the device manager to the calling device driver. Any driver or file manager must link to the specified device manager before calling it.

Parameters

`dev_name`	Contains the device name of the device manager. For DUXMan the device name is `dux`.
`dvmp`	Points to the address where the device manager handle is returned

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

`EOS_BMODE`	Returned if the caller fails to link to the device manager

Indirect Errors

`_os_link()`

Called By

Any software running in system state.

See Also

`_os_dvm_unlink()`

_os_dvm_unlink()

Unlinks from a Device Manager

Syntax

```
#include <dvm.h>
error_code _os_dvm_unlink(
     char         *dev_name,
     void         *dvmp);
```

Libraries

`os_dvm.l`

Description

`_os_dvm_unlink()` unlinks from and de-initializes a device manager such as DUXMan. This function is required when the sub-system software no longer uses the specified device manager.

Parameters

`dev_name`	Contains the device name of the device manager. For DUXMan the device name is `dux`.
`dvmp`	Points to the device handle returned by `_os_dvm_link()`

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

`EOS_BMODE`	Returned if the device fails to unlink the device manager

Indirect Errors

`_os_link()`

Called By

Any software running in system state.

See Also

`_os_dxm_pid_abtsect()`

_os_dxm_flush()

Disables All Active PID Streams

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_flush(void *dxmsp);
```

Libraries

```
os_dxm.l
```

Description

_os_dxm_flush() disables all active PID streams and clears their global flags. This function is necessary when SPF is switching MPEG2 programs or when the data input path is closed. This function also flushes the PAT cache buffer.

Parameters

dxmsp	Points to the device handle returned by _os_dvm_link()

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

EOS_NOTRDY	Returned when the receive queue for this path is empty

Called By

SPF drivers

See Also

_os_dxm_pid_delete()
_os_dxm_pid_insert()
_os_dxm_pid_status()

_os_dxm_flush_pat()

Flushes PAT Cache

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_flush_pat(void *dxmsp);
```

Description

This function flushes the DUXMan PAT cache so a new PAT can be cached.

Parameters

`dxmsp`	Points to the device handle returned by `_os_dvm_link()`

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

SPF drivers

See Also

`_os_dxm_flush()`

_os_dxm_getstat()

DUXMan Getstat Call

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_getstat(
     void          *dxmsp,
     u_int16       gs_code,
     u_int16       prm_size,
     void          *prm_blk);
```

Libraries

`os_dxm.l`

Description

`_os_dxm_getstat()` extends DUXMan for other `getstat` calls. This is a generic function used to process device-specific calls.

Parameters

`dxmsp`	Points to the device handle returned by `_os_dvm_link()`
`gs_code`	Device-specific `getstat` code defined in `<dxm_pblk.h>` or hardware layer header file `<dxm_port.h>`
`prm_size`	Contains the size of the `getstat` parameter block
`prm_blk`	Points to the `getstat` parameter block

Non-Fatal Errors

`E_UNKSVC`	`gs_code` is undefined

Called By

Any driver

See Also

`_os_dxm_setstat()`

_os_dxm_pid_abtsect()

Abort GetSect Call

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_pid_abtsect(
     void         *dxmsp,
     u_int32      sect_handle);
```

Description

This function aborts an existing `getsect` call. `sect_handle` specifies which call to abort.

Parameters

`dxmsp`	Points to the device handle returned by `_os_dvm_link()`
`sect_handle`	Section handle returned by `_os_dxm_pid_getsect()`

Non-Fatal Errors

OS-9 error code or `SUCCESS` (`0`) if no error occurred.

Called By

SPF drivers

See Also

`_os_dxm_pid_chgsect()`
`_os_dxm_pid_getsect()`

_os_dxm_pid_addbuf()

Adds Buffer for Private Data Output

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_pid_addbuf(
     void          *dxmsp,
     u_int16       pid,
     u_char        *buf,
     u_int32       bufsize);
```

Libraries

`os_dxm.l`

Description

`_os_dxm_pid_addbuf()` adds a buffer to an active private stream output. Any number of buffers can be added after the stream is activated.

DUXMan should keep a minimum of two buffers for data output at any time. This provides for private data output to multiple buffers and no data loss during the buffer switch of the output process.

Parameters

`dxmsp`	Points to the device handle returned by `_os_dvm_link()`
`pid`	Packet ID of the specified output stream
`buf`	Start address of the new buffer
`bufsize`	New buffer size

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

`EOS_ILLPRM`	Returned when the specified PID is not found
`EOS_QFULL`	Returned when the maximum number of active PIDs is exceeded

Called By

SPF drivers

_os_dxm_pid_chgsect()

Change Asynchronous GetSect Call

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_pid_chgsect (
     void          *dxmsp,
     u_int32       sect_handle,
     u_char        *new_valp,
     u_char        *new_maskp,
     error_code    (*_callback)());
```

Description

This function changes the header mask and value so different PSI/SI sections can be obtained. The caller can also specify a new callback to replace the existing one, or set this parameter to NULL to use the existing callback. sect_handle is a value returned by _os_dxm_pid_getsect().

Parameters

dxmsp	Points to the device handle returned by _os_dvm_link()
sect_handle	Section handle returned by _os_dxm_pid_getsect()
new_valp	64-bit header value
new_maskp	64-bit header mask
callback	Callback function

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

Called By

SPF drivers

See Also

`_os_dxm_pid_abtsect()`
`_os_dxm_pid_getsect()`

_os_dxm_pid_delete()

Deletes an Active PID Stream

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_pid_delete(
     void          *dxmsp,
     u_int16       pid,
     u_int32       *bufcnt);
```

Libraries

`os_dxm.l`

Description

`_os_dxm_pid_delete()` disables the output of the specified PID stream.

If the selected stream is not active, an error is returned.

If the stream type is private data and its output is a buffer, the current number of bytes of data in the buffer is returned in `bufcnt`. The `pid` is deleted.

After a stream is deleted, all events except some global events, registered on this stream are removed.

Parameters

`dxmsp`	Points to the device handle returned by `_os_dvm_link()`
`pid`	Contains the packet ID of the specified output stream
`bufcnt`	Contains the number of bytes of the private data output stream copied into the current buffer

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

`EOS_ILLPRM`	Returned when the specified PID is not found
`EOS_QFULL`	Returned when the maximum number of active PIDs is exceeded

Called By

SPF drivers

See Also

`_os_dxm_flush()`
`_os_dxm_pid_insert()`
`_os_dxm_pid_status()`

_os_dxm_pid_event()

Registers or Removes an Event

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_pid_event(
     void          *dxmsp,
     u_int16       pid,
     u_int16       ev_id,
     u_int32       ev_flag,
     void          *ev_prmblk,
     error_code    (*ev_handler)(),
     void          *ev_hdlstat);
```

Libraries

```
os_dxm.l
```

Description

`_os_dxm_pid_event()` registers or removes a specific event in the specified stream. When the registered event occurs, DUXMan dispatches and calls the callback function `ev_handler()`. Within this callback function, the caller can remove the event or delete the associated PID by calling `_os_dxm_pid_event()` or `os_dxm_pid_delete()`.

There are two kinds of events: global and non-global. The system software can register a global event when the stream PID is not known, and the PID associated with the stream is not yet active. A non-global event is registered only if the stream PID is known and after the stream has been activated.

Global events can also be registered as non-global events when the stream PID is known and the stream is activated.

When the buffer-full event `EV_BUFFER_FULL` occurs, DUXMan switches output buffers. To register this event, at least one output buffer has to be available when this event is registered.

Note

DUXMan events are not standard OS-9 events.

Parameters

`dxmsp`
Points to the device handle returned by `_os_dvm_link()`

`pid`
Packet ID of the specified output stream. If the event is global and this PID is inactive, this parameter is ignored. If the event is non-global or global and this `pid` is active, the event is registered as a non-global event.

`ev_id`
Contains two pieces of information: whether this is a global event, and the event ID. Its format is shown in the following figure.

Figure 2-1 `ev_id` Format

Global Event Indicato	0	0...................	Event ID
15	14	13.................8	7...................

When bit 15 of the `ev_id` parameter is set, it indicates that this is a global event.

The current version of DUXMan supports the following event types:

Table 2-2 PID Event IDs

ev_id	Global Event Indicator	Description
`EV_PCR`	global/ non-global	Program Clock Reference (PCR) detected
`EV_PCR_DISCONTINUITY`	global/ non-global	Discontinuous PCR detected
`EV_A_PTS`	global/ non-global	Audio PTS detected
`EV_V_PTS`	global/ non-global	Video PTS detected
`EV_A_DTS`	global/ non-global	Audio DTS detected
`EV_V_DTS`	global/ non-global	Video DTS detected
`EV_BUFFER_FULL`	non-global	Private data buffer full detected
`EV_PAYLOAD_UNIT_ START_INDICATOR`	non-global	Payload Unit Start Indicator detected
`EV_TRANSPORT_PKT_LOST`	non-global	Transport packets are lost
`EV_RANDOM_ACCESS_ ENTRY_POINT`	non-global	Random Access Entry Point is found in the stream

If a registered event occurs, DUXMan performs some pre-processing and dispatches its event-handling callback function to do the necessary processing. This event-handling callback function is defined by the caller and declared as the following:

```
error_code ev_handler(u_int16 pid,
   u_int16 ev_id, void *ev_prmblk)
```

The `ev_prmblk` points to the event parameter block allocated by the caller. For each different event type, there is a structure defined in `duxman.h` to be the parameter block structure passed through the event-handling callback function. Refer to `duxman.h` for more details.

It is possible to have multiple registrations on the same event at the same time. The event-handling callback functions are dispatched one by one according to the registration order.

`ev_flag` One of three possible values as defined in `duxman.h`:

`EV_REMOVE:` removes the event

`EV_ITERATE:` responds to the event every time it occurs

`1~0xFFFFFFFE` responds to the event the indicated number of times

`ev_prmblk` Event parameter block allocated by the caller. This block is filled with related information by DUXMan and is passed back to the caller by the event-handling callback function. The event parameter block structures for all events are defined in `duxman.h`.

Note

It is the caller’s responsibility to use the correct parameter block structure.

`ev_handler`

Points to the event-handling callback function defined by the caller. This function is called by DUXMan when the specified event occurs.

`ev_hdlstat`

Points to the static or global storage in the caller’s sub-system. The caller can get the sub-system’s global by using the `get_static()` call.

If the target processor is a 68K series, the header file `MWOS/SRC/DPIO/DEFS/defconv.h` must be included to use this call. The library `cpu.l` must be linked as well.

Non-Fatal Errors

OS-9 error code or `SUCCESS` `(0)` if no error occurred.

`EOS_ILLPRM`

Returned when the specified PID is not found

`EOS_QFULL`

Returned when the maximum number of active PIDs is exceeded

Called By

SPF and MPFM drivers

See Also

```
_os_dxm_pid_delete()
_os_dxm_flush()
_os_dxm_pid_insert()
_os_dxm_pid_status()
```

_os_dxm_pid_getpsi()

Get the PSI Tables

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_pid_getpsi(
     void         *dxmsp,
     u_int16      pid,
     u_char       table_id,
     u_char       table_rev,
     u_char       *buf,
     u_int32      *bufsize);
```

Libraries

`os_dxm.l`

Description

`_os_dxm_pid_getpsi()` gets the most recently updated PSI Table into the buffer.

If the buffer size is smaller than the requested PSI table size, DUXMan will copy as much as possible.

If the specified table is not in the specified stream, this call times out. The time-out length is determined by the DUXMan implementation.

DUXMan caches the latest version of PAT for the incoming stream. It does not update for the same version. You can force DUXMan to get the most recent PAT regardless of its version number by calling `_os_dxm_flush()`, which may be necessary when the channel is changed.

Parameters

`dxmsp`	Points to the device handle returned by `_os_dvm_link()`
`pid`	ID of the stream containing the requested PSI table

`table_id`	Table ID of the PSI table defined in the ISO/IEC 13818-1 DIS specification. For PAT, CAT, and PMT, the `table_id` is 0, 1, and 2 respectively.
`table_rev`	Bits 0 through 4 contain the version number of the requested table. Bit 7 indicates if version number should be used. Set it if you want DUXMan to use the version number, clear it if you do not. DUXMan gets the next available version in the stream if this bit is cleared.
`buf`	Buffer start address
`bufsize`	Buffer size as input. The actual number of bytes copied is returned here.

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

`EOS_NOTRDY`	Returned when the receive queue for this path is empty
`EOS_REQ_TIMEOUT`	Returned when the specified table is not in the stream
`EOS_DEVBSY`	Returned when multiple processes try to access the same table on the same stream at the same time
`EOS_ILLPRM`	Returned when the specified PID is not found
`EOS_QFULL`	Returned when maximum number of active PIDs is exceeded

Called By

SPF drivers

_os_dxm_pid_getsect()

Get Section Data Asynchronously

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_pid_getsect (
     void         *dxmsp,
     u_int16      pid,
     u_char       *head_valp,
     u_char       *head_maskp,
     error_code   (*_callback)(),
     void         *_callback_stat,
     u_int32      *sect_handle);
```

Description

This function accepts a pointer to DUXMan (through the `_os_dvm_link()` call), a PID within which the requested data is carried, a 64-bit header value, a 64-bit header mask, a callback function and corresponding static storage, and a pointer to store the handle.

DUXMan fills in the handle so later calls to change the mask or abort the call can be referenced by the handle. The 64-bit mask indicates in which fields in a PSI/SI section header the caller has interest. A zero in a field means the caller does not care about this field. The 64-bit header value indicates to DUXMan what values these fields should be if the corresponding fields in the mask are not zero. The callback function is called by DUXMan when the requested table information is available.

Parameters

`dxmsp`	Points to the device handle returned by `_os_dvm_link()`
`pid`	Packet ID of the data
`head_valp`	64-bit header value
`head_maskp`	64-bit header mask
`_callback`	Callback function
`_callback_stat`	Static storage of caller
`sect_handle`	Handle for future reference

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

SPF drivers

See Also

`_os_dxm_pid_abtsect()`
`_os_dxm_pid_chgsect()`

_os_dxm_pid_insert()

Selects a Specific PID
Stream to Output

Syntax

```
#include <DAVID/duxman.h>
#include <mpeg.h>
error_code _os_dxm_pid_insert(
     void          *dxmsp,
     u_int16       pid,
     u_int16       istr_type,
     u_int16       ostr_type,
     u_char        *buf,
     u_int32       bufsize);
```

Libraries

`os_dxm.l`

Descripton

`_os_dxm_pid_insert()` activates the output of the stream with the specified PID.

If the selected stream type is audio or video, the output is routed directly to the corresponding MPEG decoder.

If the selected stream type is private data, the output is a high-speed RAM.

Parameters

`dxmsp`	Points to the device handle returned by `_os_dvm_link()`
`pid`	Packet ID of the specified output stream
`istr_type`	PCR PID indicator and the stream type. The format of `istr_type` is shown in the following figure:

Figure 2-2 `istr_type` Format

PCR_PI Flag	0	0................(	Output Strean Type
15	14	13................8	7..................C

The least significant 8 bits indicate the input stream type containing all the possible stream type values defined in the ISO/IEC 13818 DIS MPEG specification. Typical values are:

```
0x01MPEG_1_VIDEO
0x02MPEG_2_VIDEO
0x03MPEG_1_AUDIO
0x04MPEG_2_AUDIO
0x05PVT_13818_1
```

For more definitions refer to the `mpeg.h` file.

Bit 15 indicates whether the stream contains PCR information or not. It is set to 1 when the stream contains PCR information. This value is defined as `PCR_PID_FLAG` in `duxman.h`.

`ostr_type`

For video and audio streams, this field is ignored. Any value is fine. For private streams, it contains two pieces of information, the `payload_unit_start_indicator` flag, and output stream types for private data. The format of `ostr_type` is shown in the following figure.

Figure 2-3 `ostr_type` Format

payload_ unit_start_ ndicator	0	0................(	Output Strean Type
15	14	13................8	7..................C

The possible output stream types are:

`O_TYPE_TRANS:` output the entire transport stream of the specified PID

`O_TYPE_TP_PL:` output only the payload part of the specified transport stream

`O_TYPE_PES_PL:` output only the payload part of the PES packets

`O_TYPE_NULL:` no output is necessary

`O_TYPE_SECT:` section data output

`O_TYPE_UNKNOWN:` output stream type not known or not important (such as audio/video stream)

Setting bit 15 of `ostr_type` indicates to DUXMan to start receiving data at the packet having the `payload_unit_start_indicator` bit set. Otherwise DUXMan receives the data at the first incoming packet.

For more information refer to the `duxman.h` file.

`buf`
Start address of the buffer to output the selected stream. Used only for private data streams. If a stream is routed to an output device such as a hardware decoder, this parameter is not used. Before the inserted stream fills all the buffers, the caller can add additional buffers by calling `_os_dxm_pid_addbuf()` to provide more output buffers, or delete the stream by calling `_os_dxm_pid_delete()`.

`bufsize`
Size of the output buffer. Used only for private data streams.

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

`EOS_ILLPRM`	Returned when the specified PID is not found
`EOS_QFULL`	Returned when the maximum number of active PIDs is exceeded

Called By

SPF driver

See Also

`_os_dxm_pid_delete()`
`_os_dxm_flush()`
`_os_dxm_pid_status()`

_os_dxm_pid_status()

Gets the Status of a PID stream

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_pid_status(
     void          *dxmsp,
     u_int16       pid,
     pid_status_blk *blk);
```

Libraries

`os_dxm.l`

Description

`_os_dxm_pid_status()` returns information about the specified PID stream.

This information includes:

- The number of times the buffer has been filled
- The number of bad packets received for this stream
- The total number of transport packets received
- The total number of transport packets lost

Parameters

`dxmsp`	Points to the device handle returned by `_os_dvm_link()`
`pid`	Packet ID of the specified output stream
`blk`	Points to the PID status block structure. This block structure is defined in `duxman.h`.

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

`EOS_ILLPRM`	Returned when the specified PID is not found

Called By

SPF drivers.

See Also

`_os_dxm_pid_delete()`
`_os_dxm_pid_event()`
`_os_dxm_flush()`
`_os_dxm_pid_insert()`

_os_dxm_setstat()

DUXMan Setstat Call

Syntax

```
#include <DAVID/duxman.h>
error_code _os_dxm_setstat(
     void          *dxmsp,
     u_int16       ss_code,
     u_int16       prm_size,
     void          *prm_blk);
```

Libraries

os_dxm.l

Description

_os_dxm_setstat() extends DUXMan for other setstat calls. This is a generic function used to process device-specific calls.

Parameters

dxmsp	Points to the device handle returned by _os_dvm_link()
ss_code	Device-specific setstat code. ss_code is defined in <dxm_pblk.h> or <dxm_port.h> if any are supported.
prm_size	Size of the setstat parameter block
prm_blk	Points to the setstat parameter block

Non-Fatal Errors

E_UNKSVC	ss_code is undefined

Called By

_os_dxm_getstat()

Chapter 3: Porting DUXMan

This chapter describes the tasks needed to port DUXMan to your target environment.

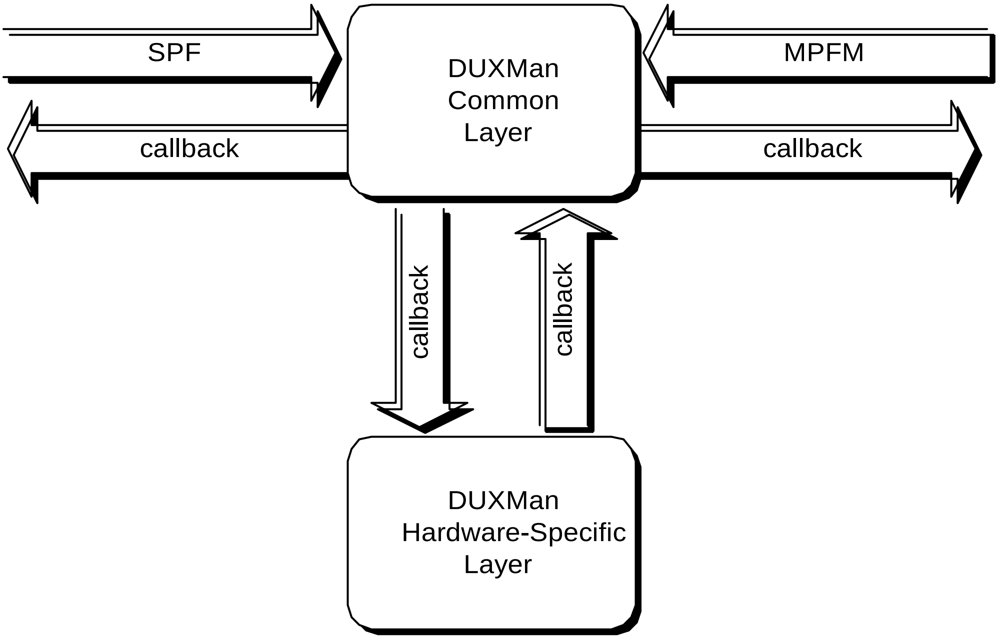
The following sections are included in this chapter:

- **DUXMan Software Layers**
- **DUXMan Key Data Structures**
- **DUXMan Functional Implementations**
- **Hardware-Specific Layer Functions**
- **Hardware-Specific Layer Functions**

DUXMan Software Layers

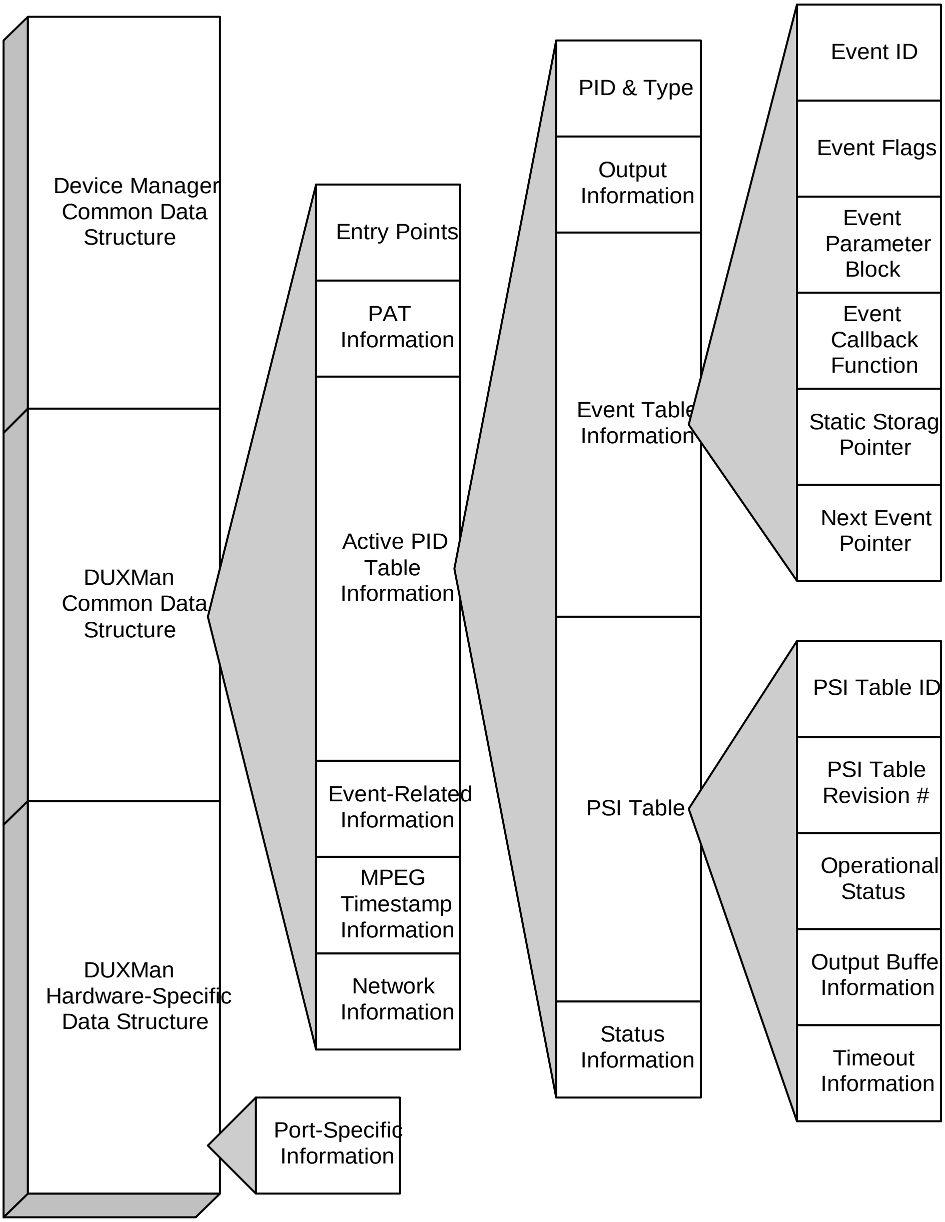
DUXMan software has two layers: the common layer and the port-specific or hardware-specific layer as shown in **Figure 3-1**.

Figure 3-1 DUXMan Software Layers



The Static Storage Data Structure `dxm_stat` of DUXMan is depicted in **Figure 3-2**.

Figure 3-2 DUXMan Static Storage Data Structure



DUXMan Key Data Structures

This section contains definitions of some key data structures in DUXMan. These structures are defined in `dxm_sys.h`.

Table 3-1 DUXMan Data Structures

Data Structure	Description
`_dxm_stat`	DUXMan Driver Static Storage Structures
`_sib`	Stream Information Block (SIB) Structure
`_eqe`	Event Queue Entry Structure
`_tqe`	PSI Table Entry Structure
`_sqe`	SQE Structure

_dxm_stat

DUXMan Driver Static Storage Structures

Syntax

```
Typedef struct _dxm_stat {

   /* device manger common data structure */

   dvm_stat v_dvm;         /* device manager common structure */
   u_int32  v_nof_entries; /* number of entry points */
   error code  (*v_dxm_register)();
   error_code  (*v_dxm_pid_insert)();
   error_code  (*v_dxm_pid_delete)();
   error_code  (*v_dxm_pid_add_buf)();
   error_code  (*v_dxm_pid_getpsi)();
   error_code  (*v_dxm_pid_status)();
   error_code  (*v_dxm_pid_event)();
   error_code  (*v_dxm_flush)();
   error_code  (*v_dxm_getstat)();
   error_code  (*v_dxm_setstat)();
   error_code  (*v_dxm_pid_getsect)();
   error_code  (*v_dxm_pid_chgsect)();
   error_code  (*v_dxm_pid_abtsect)();
   error_code  (*v_dxm_flush_pat)();

   /* PAT information */

   u_char *v_pat_buf;       /* PAT buffer pointer */
   u_int32 v_pat_bufcnt;    /* PAT buffer data count */
   u_int32 v_pat_bufsize;   /* PAT buffer size */
   u_char  v_pat_avail_ver; /* available version */
   u_char  v_pat_bufovf;    /* buffer overflow flag */
   u_int16 rsvd1;           /* reserved */


   /* Active PID Table information */

   u_int16 v_sib_max;      /* maximum number of SIBs in table */
   u_int16 v_sib_count;    /* number of active SIBs in table */
   /* Active PID Table information continued */

   u_int16 v_sib_tbmax;    /* maximum number of PSI table
                           /* entries in each SIB */
   u_int16 rsvd2;          /* reserved */
   sib *v_sib_table;       /* SIB table (APT) start address */
   sib *v_cur_sib;         /* most recently activated SIB */
   sib *v_aud_sib;         /* active audio SIB */
   sib *v_vid_sib;         /* active video SIB */
   sib *v_pcr_sib;         /* active PCR SIB */

   /* Event related information */

   eqe v_ev_globtab[EV_GLOB_MAX+1];
                           /* global event table */
   u_int32 v_ev_max;       /* maximum number of free event
                           /* in the free event pool */
   eqe *v_ev_pool;         /* free event pool start address */

   /* MPEG time stamp information */

   pcrctxt v_pcr;          /* current PCR context */
   u_char v_pts[12];       /* PTS buffer */

   /* Network transmission information */

   u_int32 v_sync_lost;    /* number of times sync is lost */
   u_int32 v_fifo_full;    /* number of times fifo is full */
   u_int32 v_dev_max;      /* maximum number output devices */
   deve *v_dev_list;       /* output device list */

   /* Port specific information */
   #ifdef DRVR_HW_SPECIFIC
   DRVR_HW_SPECIFIC
   #endif

} dxm_stat;
```

Description

_dxm_stat defines the driver static storage structure.

If required, DRVR_HW_SPECIFIC is defined in the hardware layer header file (dxm_port.h).

_sib

Stream Information Block (SIB) Structure

Syntax

```
typedef struct _sib
{
   u_int16 pid;            /* current stream PID value*/
   u_int16 ostr_type;      /* output stream type */
   u_int16 istr_type;      /* input stream type */
   u_int16 op_status;      /* operational status of current
                           /* SIB */
   u_char *obuf;           /* output buffer pointer */
   u_char *sbuf;           /* secondary output buffer pointer */
   u_int32 obuf_size;      /* output buffer size */
   u_int32 sbuf_size;      /* secondary output buffer size */
   u_int32 obuf_cnt;       /* output buffer data count*/
   u_int32 continuity_cnt; /* PID continuity count */

   eqe ev_table[EV_MAX + 1];  /* event queue array */

   u_int16 ev_count;       /* event queue count */
   u_int16 ev_id;          /* event id */
   tqe tb_queue;           /* PSI table queue start address */
   u_int16 use_cnt;        /* number of processes using this SIB */
   u_int16 tb_cnt;         /* number of table entries used */
   sqe *stb_queue;         /* sub-table queue start address */
   u_int16 stb_cnt;        /* number of sub-table entries used*/
   u_int16 resved;         /* reserved */
   u_int32 buf_full;       /* # times current PID buffer
                           /* is full */
   u_int32 bad_pkt;        /* # bad transport packets received */
   u_int32 rcvd_pkt;       /* total # of transport packets
                           /* received */
   u_int32 lost_pkt;       /* total # of transport packets
                           /* lost */
   #define HW_RESERVED_BYTES 32
   union {
   #ifdef SIB_HW_SPECIFIC
       SIB_HW_SPECIFIC      /* hardware-specific part of the
                            /* SIB */
   #endif
      u_int32 rsvd[HW_RESERVED_BYTES/4]
   }reserved;

} sib, *Sib;
```

Description

This is the structure used to store PID-specific stream information.

_eqe

Event Queue Entry Structure

Syntax

```
typedef struct _eqe
{
   u_int16 ev_first_flag;  /* first event flag
                           /* (used for PCR_DISC) */
   u_int16 ev_id;          /* event identifier (0~MAXEVENT_ID) */
   u_int32 ev_flag;        /* event handling iteration flag */

   void *ev_prmblk;        /* parameter block address */

   error_code(*ev_handle)(u_int32, u_int32, void *);
                           /* pointer to this entry's event
                           /* handler */

   void *ev_hdlstat;       /* statics for calling ev_handle() */

   struct _eqe *ev_next;   /* next event entry */
} eqe, *Eqe;
```

Description

This structure stores information related to each event registration request.

_tqe

PSI Table Entry Structure

Syntax

```
Typedef struct _tqe
{
   u_int16 table_id;        /* PSI table ID */
   u_char table_rev;        /* PSI table revision number */
   u_char op_flag;          /* table operation flag */
   u_char *buf;             /* PSI table output buffer address */
   u_int32 bufsize;         /* PSI table buffer size */
   u_int32 bufcnt;          /* PSI table output buffer count */
   u_int32 timeout_param;   /* timeout parameter */
   u_int32 rsvd[2];         /* reserved */
} tqe, *Tqe;
```

Description

This structure defines the PSI table entry structure.

_sqe

SQE Structure

Syntax

```
typedef struct _sqe {
   u_int16 s_sync_word;    /* Flag and also error */
                           /* checking*/

#define SECT_SYNC_WORD  0x465A
                           /* char string "FZ" */
   u_int16 s_pid;          /* PID for this sub-table */
   void *s_sib;            /* Pointer to the related */
                           /* SIB */
   u_char *s_hd_valp;      /* Head value pointer */
                           /* (64 bits) */
   u_char *s_hd_maskp;     /* Head mask pointer */
                           /* (64 bits) */
   error_code (*s_callback)(void *stat, void *buf,
               u_int32 sect_handle, u_int32 pid);
                           /* Callback fuction pointer */
   void *s_cb_stat;        /* Caller's global storage */

   /*-------------------------------------------------*/
   /* The following fields are derived from the above */
   /* fields. But define them to avoid getting them   */
   /* every time they are used                        */
   /*-------------------------------------------------*/
   u_int32 s_hd_val1;      /* First 4 bytes of header */
                           /* val */
   u_int32 s_hd_val2;      /* second 4 bytes of header */
                           /* val*/
   u_int32 s_hd_mask1;     /* First 4 bytes of header */
                           /* mask*/
   u_int32 s_hd_mask2;     /* second 4 bytes of header */
                           /* mask*/
   u_char s_table_id;      /* table ID requested */
   u_char s_ver_num;       /* version number requested */
#define SECT_VER_ANY 0xFF  /* Version don't care */
   u_char s_sect_num;      /* section number requested */
   u_char s_cur_next;      /* current/next indicator */
   u_int16 s_subtab_id;    /* sub-table ID requested */
   u_int16 s_entry_index;  /* index of this entry in */
```

```
                           /* a SIB*/

   /*-------------------------------------------------*/
   /* Some actual parameters from the data stream     */
   /*-------------------------------------------------*/
   u_int16 s_sect_length;  /* sect_length from the */
                           /* stream */
   u_char s_real_sect_num; /* Actual sect num in stream */
   u_char s_real_last_num; /* Last sect num in the */
                           /* stream */
   u_char s_real_ver;      /* Actual ver in the stream */
   u_char s_resved1;       /* reserved */
   u_int16 s_sesved2;      /* reserved */

   /*------------------------------*/
   /* old header val and mask      */
   /*------------------------------*/
   u_char s_old_val[8];    /* old head value */
   u_char s_old_mask[8];   /* old head mask */

   /*------------------------------*/
   /* change flags                 */
   /*------------------------------*/
   u_int32 s_tab_chg_flag:1;    /* table_id changed when =1 */
   u_int32 s_subtab_chg_flag:1;
                              /* sub-table_id changed when =1 */
   u_int32 s_ver_chg_flag:1;    /* ver num  changed when =1 */
   u_int32 s_cn_chg_flag:1;     /* cur_next changed flag */
   u_int32 s_sec_num_chg_flag:1;/* sect number changed flag */


   /*------------------------------*/
   /* field valid flags            */
   /*------------------------------*/
   u_int32 s_subtab_flag:1;     /* sub-table num valid when=1 */
   u_int32 s_cn_flag:1;       /* cur/next valid flag:mask */
   u_int32 s_sect_num_flag:1;   /* sect num valid flag:mask */
   u_int32 s_rsvd_flag:24;    /* reserved */
} sqe, *Sqe;
```

Description

Because of the number of fields in the section data syntax, this data structure defines all these corresponding fields for convenience of use. The caller provides the PID, section header value, section header mask, callback function, and caller static storage. DUXMan maintains one such data structure for each valid call in the system and provides a handle to the caller. The handle is actually the pointer to the related structure. The field `s_sync_word` is used to check for the faked handle. The section header and header mask are 8-byte strings that determine which fields are used to get requested section tables. Only the fields that are non-zero in the header mask are used. The actual values for these fields are the corresponding fields in the sections header.

For each PID, the driver allows only a limited number of tables to be processed simultaneously. This is determined by `v_sib_tbmax` defined in the data structure `_dxm_stat`. It is setup in the DUXMan descriptor.

DUXMan Functional Implementations

Table 3-2, summarizes the DUXMan common layer functions. These functions call the hardware layer code and are not typically modified for a port. To port DUXMan, however, it is best to know how the common layer functions work.

Table 3-2 DUXMan Common Layer Functions

Function	Description
`dvm_init()`	Initializes DUXMan and Hardware Layer
`dvm_term()`	De-initializes DUXMan and Hardware Layer
`dxm_flush()`	Flushes all PIDs in the Active PID Table
`dxm_flush_pat()`	Flush PAT Cache
`dxm_getstat()`	DUXMan getstat
`dxm_pid_abtsect()`	Abort the dxm_pid_getsect Call
`dxm_pid_addbuf()`	Adds Buffer to an Active PID Stream
`dxm_pid_chgsect()`	Change the Mask and Value
`dxm_pid_delete()`	Deletes a PID from the Active PID Table
`dxm_pid_event()`	Adds or Removes an Event Handler
`dxm_pid_getpsi()`	Gets the PSI Table
`dxm_pid_getsect()`	Get PSI/SI Sections
`dxm_pid_insert()`	Inserts a PID into the Active PID Table

Table 3-2 DUXMan Common Layer Functions (continued)

Function	Description
`dxm_pid_status()`	Get Status for a Specific PID Stream
`dxm_setstat()`	DUXMan setstat Function

dvm_init()

Initializes DUXMan and Hardware Layer

Syntax

```
#include <dxm_sys.h>
error_code dvm_init(dvm_stat *dvmsp);
```

Description

`dvm_init()` does the following:

- Allocates internal memory from system RAM or data module:

  * allocate memory for the maximum number of SIBs
  * allocate memory for an event array with the maximum number of event entries for each SIB
  * allocate memory for PAT table buffers

- Initializes function entry points
- Initializes SIB structures
- Initializes other DUXMan global variables
- Calls a hardware-specific function such as `hw_dxm_init()` to do the port-specific initialization and install the interrupt service routines

Parameters

`dvmsp`	Points to DUXMan global storage

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`_os_dvm_link()`

dvm_term()

De-initializes DUXMan and Hardware Layer

Syntax

```
#include <dxm_sys.h>
error_code dvm_term(dvm_stat *dvmsp);
```

Description

dvm_term() does the following:

- Calls hardware specific functions such as hw_dxm_term() for the port-specific de-initialization, including removing installed interrupt service routines
- Frees allocated internal memory
- Clears some of the DUXMan global variables

Parameters

dvmsp	Points to DUXMan global storage

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

Called By

_os_dvm_unlink()

dxm_flush()

Flushes all PIDs in the Active PID Table

Syntax

```
#include <dxm_sys.h>
error_code dxm_flush(dxm_stat *dxmsp);
```

Description

dxm_flush() reinitializes the TDIC and clears all active PID streams and other data structures and flags, including flushing the PAT cache.

This function calls the hardware layer function hw_dxm_flush().

Parameters

dxmsp	Points to DUXMan global storage

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

Called By

_os_dxm_flush()

dxm_flush_pat()

Flush PAT Cache

Syntax

```
#include <dxm_sys.h>
error_code dxm_flush_pat(dxm_stat *dxmsp);
```

Description

This function resets the PAT buffer count and calls the hardware layer function `hw_dxm_flush_pat()`.

Parameters

`dxmsp`	Points to DUXMan global storage

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`_os_dxm_flush_pat()`

See Also

`dxm_flush()`

dxm_getstat()

DUXMan getstat

Syntax

```
#include <dxm_sys.h>
error_code dxm_getstat(
     dxm_stat     *pbp);
```

Parameter Block

```
typedef struct _dxm_gs_ss_pb
{
   u_int16 scode;           /* function code */
   u_int16 prm_size;        /* parameter size */
   void *prm_blk;           /* pointer to the scode-*/
                            /* specific parameter block */
} dxm_gs_ss_pb, *Dxm_gs_ss_pb;
```

Description

dxm_getstat() passes the call to the hardware-specific getstat function hw_dxm_getstat() to process a specifically defined DUXMan getstat call.

Parameters

dxmsp	Points to DUXMan global storage
pbp	Points to the parameter block

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

Called By

_os_dxm_getstat()

dxm_pid_abtsect()

Abort the dxm_pid_getsect Call

Syntax

```
#include <dxm_sys.h>
error_code dxm_pid_abtsect(
     dxm_stat     *dxmsp,
     dxm_pid_abtsect_pb *pb);
```

Parameter Block

```
typedef struct _dxm_pid_abtsect_pb {
   u_int32 sect_handle;    /* handle to sub-table */
                           /* entry */
} dxm_pid_abtsect_pb, *Dxm_pid_abtsect_pb;
```

Description

For PAT, this function verifies the handle is valid, resets the sync word to 0, and sets the version number to SECT_VER_ANY and the callback function to NULL.

For any other table, this function verifies the handle, calls hardware layer function `hw_pid_abtsec()`, and resets all related fields in the `sqe` data structure. If this is the last table for the SIB, remove the SIB.

Parameters

`dxmsp`	Points to DUXMan global storage
`pb`	Points to the parameter block

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`_os_dxm_pid_abtsect()`

See Also

`dxm_flush_pat()`
`dxm_pid_chgsect()`
`dxm_pid_getsect()`

dxm_pid_addbuf()

Adds Buffer to an Active PID Stream

Syntax

```
#include <dxm_sys.h>
error_code dxm_pid_add_buf(
     dxm_stat     *dxmsp,
     dxm_add_buf_pb *pbp);
```

Parameter Block

```
typedef struct _dxm_add_buf_pb
{
   u_int16 pid;     /* PID value      */
   u_int16 rsvd;    /* reserved       */
   u_char *buf;     /* buffer address */
   u_int32 bufsize; /* buffer size    */
} dxm_add_buf_pb, *Dxm_add_buf_pb;
```

Description

`dxm_pid_add_buf()` attaches a buffer to an active PID stream which carries private data.

This function calls the hardware layer function `hw_add_buf()`.

After a `dxm_pid_insert()` call, the caller can issue one or more `dxm_pid_addbuf()` calls to add buffers for reading the private data stream.

DUXMan maintains the currently active output buffer and a secondary output buffer that links to as many buffers as the caller has added.

Parameters

`dxmsp`	Points to DUXMan global storage
`pbp`	Points to the parameter block

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`_os_dxm_pid_addbuf()`

dxm_pid_chgsect()

Change the Mask and Value

Syntax

```
#include <dxm_sys.h>
error_code dxm_pid_chgsect(
     dxm_stat     *dxmsp,
     dxm_pid_chgsect_pb *pbp);
```

Parameter Block

```
typedef struct _dxm_pid_chgsect_pb {
   u_int32 sect_handle;    /* handle to sub-table entry */
   u_char *header_valp;    /* New header value pointer */
   u_char *header_maskp;   /* New header mask pointer */
   error_code (*sect_callback) (void *callback_stat,
      void *buf, u_int32 sect_handle, u_int32 pid);
                           /* callback function */
} dxm_pid_chgsect_pb, *Dxm_pid_chgsect_pb;
```

Description

After error-checking, the function saves the new parameters into the sqe structure. For PAT, update the section version number and fill the rest of the header into the data structure. If the requested version is already available, call the pat_callback() function to send the table to the caller.

For other tables, check each field in the header and identify the changes. Then call the hardware layer function hw_pid_chgsec().

Parameters

dxmsp	Points to DUXMan global storage
pbp	Points to the parameter block

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

Called By

`_os_dxm_pid_chgsect()`

See Also

`dxm_flush_pat()`
`dxm_pid_abtsect()`
`dxm_pid_getsect()`

dxm_pid_delete()

Deletes a PID from the Active PID Table

Syntax

```
#include <dxm_sys.h>
error_code dxm_pid_delete(
     dxm_stat     *dxmsp,
     dxm_delete_pid_pb *pbp);
```

Parameter Block

```
typedef _dxm_delete_pid_pb
{
   u_int16 pid;          /* PID value */
   u_int16 rsvd;         /* reserved */
   u_int32 bufcnt;       /* the byte count of data that */
                         /* has been put in the buffer */
} dxm_delete_pid_pb, *Dxm_delete_pid_pb;
```

Description

dxm_pid_delete() searches through the Active PID Table to find and delete the specified PID. If this PID stream is a private-data stream being output to a buffer, this function writes the number of bytes in the buffer to bufcnt in the parameter block.

This function calls the hardware layer function hw_pid_delete().

Parameters

dxmsp	Points to DUXMan global storage
pbp	Points to the parameter block

Non-Fatal Errors

OS-9 error code or SUCCESS	(0) if no error occurred.
EOS_ILLPRM	Returned when the specified PID is not found

Called By

`_os_dxm_pid_delete()`

dxm_pid_event()

Adds or Removes an Event Handler

Syntax

```
#include <dxm_sys.h>
error_code dxm_pid_event(
     dxm_stat      *dxmsp,
     dxm_pid_event_pb *pbp);
```

Parameter Block

```
typedef struct _dxm_pid_event_pb
{
   u_int16 pid;            /* PID value */
   u_int16 ev_id;          /* event identifier */
   u_int32 ev_flag;        /* event handling iterating flag */
     void *ev_prmblk;      /* event handler parameter block */
   error_code (*ev_handle)(u_int32 pid,
      u_int32 ev_id,
      void *ev_prmblk);    /* event handler function */
      void *ev_hdlstat;    /* statics for calling */
                           /* ev_handle() */
} dxm_pid_event_pb, *Dxm_pid_event_pb;
```

Description

`dxm_pid_event()` adds or deletes an event entry to or from the event queue.

This function calls the hardware layer functions `hw_insert_event()` and `hw_remove_event()`.

When DUXMan dispatches a call to an event handler, the first parameter passed is always the stream PID that relates to the event.

The second parameter is the event ID, which is defined according to the different events.

The third parameter is a parameter block pointer, which is also event specific. Refer to `duxman.h` for the parameter block structure definitions.

Parameters

`dxmsp` Points to global DUXMan structure

Inside the parameter block `pbp`, the following parameters are defined:

`pid` Identifies the stream PID for the event

`ev_id` Identifies the type of event and indicates if this event is a global event

DUXMan first checks the global event indicator (bit 15). If it is a global event, DUXMan stores it in the global event table and inserts the event when its correspondent stream becomes active.

Currently, the following types of events are supported in DUXMan:

Event 0 - PCR found
Event 1 - PCR discontinuity found
Event 2 - Audio PTS found
Event 3 - Video PTS found
Event 4 - Audio DTS found
Event 5 - Video DTS found
Event 6 - Stream output buffer is full
Event 7 - Payload_unit_start_indicator set
Event 8 - Transport packet lost
Event 9 - Random access entry point is found

According to the event type indicated by `ev_id`, after the `dxm_pid_event()` call, DUXMan does the following:

Locates the PID entry where the event queue is located, if the PID entry is not found an `EOS_ILLPRM` error is returned.

Checks the input `ev_flag`:

If the `ev_flag` is 0 (`EV_REMOVE`), DUXMan removes the event entry from the event queue. If this event entry cannot be found in the event queue SUCCESS is returned.

If `ev_flag` is n, DUXMan inserts this event handler entry into the event queue. When this event is detected, DUXMan dispatches its event handler and the `ev_flag` is decremented by one. After n times, the event entry is removed from the event queue.

If `ev_flag` is 0xFFFFFFFF (`EV_ITERATE`), DUXMan inserts this event entry into the event queue. Every time this event is detected, DUXMan dispatches its event handler.

`ev_handle` Is the event handler function pointer

`ev_hdlstat` Points to the caller's static storage

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

`EOS_ILLPRM` Returned if event handler is NULL, `ev_id` is out of range, or PID entry not found when adding an event

Called By

`_os_dxm_pid_event()`

dxm_pid_getpsi()

Gets the PSI Table

Syntax

```
#include <dxm_sys.h>
error_code dxm_pid_getpsi(
     dxm_stat     *dxmsp,
     dxm_pid_getpsi_pb *pbp);
```

Parameter Block

```
typedef _dxm_pid_getpsi_pb
{
   u_int16 pid;            /* PID value */
   u_char table_id;        /* table ID */
   u_char table_rev;       /* table revision number */
   u_char *buf;            /* buffer address */
   u_int32 bufsize;        /* buffer size count */
} dxm_pid_getpsi_pb, *dxm_pid_getpsi_pb;
```

Description

`dxm_pid_getpsi()` gets the most recent Program Specific Information (PSI) tables such as the Program Association Table (PAT), or the Program Map Table (PMT) into the provided buffer.

If the buffer size is smaller than the required table size, this function copies as much data as possible.

`dxm_pid_getpsi()` also stores the size of the data actually copied to the buffer in the parameter block's `bufsize` field.

The PAT is always cached by DUXMan. Other tables are not cached, so it may take more time to return to the caller.

If, after a period of time, DUXMan still cannot get the requested table, this function times out. It is the responsibility of the hardware layer to specify this time-out period.

This function calls the hardware layer function `hw_pid_getpsi()`.

Parameters

`dxmsp`	Points to DUXMan global storage
`pbp`	Points to the parameter block

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

`EOS_NOTRDY`	Returned when PAT is not available or the system is not ready
`EOS_ILLPRM`	Returned when the specified PID is not found
`EOS_QFULL`	Returned when the maximum number of active PIDs is exceeded
`EOS_DEVBSY`	Returned when multiple processes try to access the same table on the same stream at the same time

Called By

`_os_dxm_pid_getpsi()`

dxm_pid_getsect()

Get PSI/SI Sections

Syntax

```
#include <dxm_sys.h>
error_code dxm_pid_getsect
     (dxm_stat    *dxmsp,
     dxm_pid_getsect_pb *pbp);
```

Parameter Block

```
typedef struct _dxm_pid_getsect_pb {
   u_int16 pid;            /* PID value */
   u_char *header_valp;    /* 64-bit header value ptr */
   u_char *header_maskp;   /* 64-bit header mask ptr */
   error_code (*sect_callback) (void *callback_stat,
       void *buf, u_int32 sect_handle, u_int32 pid);
                           /* callback function */
   void *callback_stat;    /* caller's static storage */
   u_int32 *sect_handlp;   /* section handle pointer */
} dxm_pid_getsect_pb, *Dxm_pid_getsect_pb;
```

Description

For PAT, this function sets up all related parameters based on the input parameter. If the requested version is already available, call `sect_callback()` to send the table to the caller. Otherwise, send table when available later.

For other tables, if no SIB has been allocated for this table, get a new SIB. Find an empty space for the new table entry and call the hardware function `hw_pid_getsec()`.

Parameters

`dxmsp`	Points to DUXMan global storage
`pbp`	Points to the parameter block

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`_os_dxm_pid_getsect()`

See Also

`dxm_flush_pat()`
`dxm_pid_abtsect()`
`dxm_pid_chgsect()`

dxm_pid_insert()

Inserts a PID into the Active PID Table

Syntax

```
#include <dxm_sys.h>
error_code dxm_pid_insert
     (dxm_stat    *dxmsp,
     dxm_insert_pid_pb *pbp);
```

Parameter Block

```
typedef _dxm_insert_pid_pb
{
   u_int16 pid;            /* PID value */
   u_int16 istr_type;      /* input stream type */
   u_int16 ostr_type;      /* output stream type */
   u_int16 rsvd;           /* reserved */
   u_char *buf;            /* output buffer address */
   u_int32 bufsize;        /* output buffer size*/
} dxm_insert_pid_pb, *Dxm_insert_pid_pb;
```

Description

`dxm_pid_insert()` inserts a new PID into the active PID table.

After the PID is inserted, DUXMan takes the specified PID data from the incoming transport stream and parses it to the required layer according to the output stream type.

Audio/video data is sent directly to the decoders in its required form (Packetized Elementary Stream (PES) or elementary stream).

Private data are streamed out in the format indicated by its output stream type.

This function calls the hardware layer function `hw_pid_insert()`.

Parameters

`dxmsp` Points to DUXMan global storage

`pbp` Points to the parameter block

Inside the parameter block structure the following parameters are defined:

`pid` Contains the input stream PID value

`istr_type` Contains the input stream type. Its format is shown in the following figure.

Figure 3-3 `istr_type` Format

PCR_PID Flag	0	0..................0	Input Stream Type
15	14	13................8	7..................0

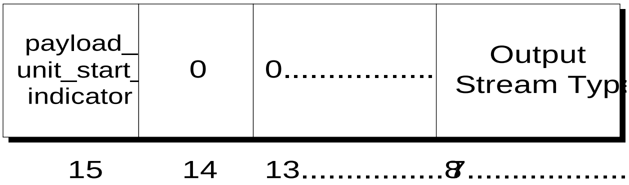
The least significant byte of the input stream type `istr_type` is the stream type defined in ISO/IEC 138180-1 DIS Specification Table 2-36. The most significant bit of this field contains the `PCR_PID` flag. This flag is set when the stream contains a PCR value.

If the input stream type has the `PCR_PID` flag set, DUXMan turns on the PCR clock recovery function.

When the input stream type is of type audio or video the stream type can be set to `O_TYPE_UNKNOWN`. When the input stream contains only PCR information, the output stream type can be set to `O_TYPE_NULL`.

`ostr_type` Contains the output stream type and `payload_unit_start_indicator` flag. Its format is shown in the following figure.

Figure 3-4 ostr_type Format



The most significant bit of this field indicates the `payload_unit_start_indicator` flag. When it is set, DUXMan starts streaming data at the first packet that has the `payload_unit_start_indicator` flag set. The least significant byte of this field indicates the output stream types.

When the output stream goes to system RAM, the output stream type must be set to one of the following values:

Symbol	**Description**
O_TYPE_TRANS	MPEG transport packet
O_TYPE_TP_PL	MPEG transport packet payload
O_TYPE_PES_PL	MPEG PES packet payload

`buf` Contains the output buffer address

`bufsize` Contains the size of the output buffer

If the output of the data goes to a buffer, and the buffer becomes full, DUXMan dispatches the `EV_BUFFER_FULL` event (if the event has been registered), and checks to see if a secondary output buffer is assigned. If a secondary buffer has been assigned, DUXMan resets this buffer to be the current output buffer and links it to a newly assigned secondary buffer, if one exists. DUXMan continues to output data until no more buffers are available or the PID is deleted.

Non-Fatal Errors

OS-9 error code or `SUCCESS`	`(0)` if no error occurred.
`EOS_QFULL`	Returned when maximum number of active PIDs exceeded

Called By

`_os_dxm_pid_insert()`

dxm_pid_status()

Get Status for a Specific PID Stream

Syntax

```
#include <dxm_sys.h>
error_code dxm_pid_status(
     dxm_stat     *dxmsp,
     dxm_pid_status_pb *pbp);
```

Parameter Block

```
typedef _dxm_pid_status_pb
{
   u_int16 pid;            /* PID value */
   u_int16 rsvd            /* reserved field */
   pid_status_blk *pidstat;
                           /* pointer to the PID’s
                           /*  net status block */
} dxm_pid_status_pb, *Dxm_pid_status_pb;

typedef _pid_status_blk {
   u_int32 buf_full;       /* # times the pid bit rate
                           /* buffer is full */
   u_int32 bad_pkt;        /* # bad transport packets */
   u_int32 rcvd_pkt;       /* # total transport packet
                           /* received  */
   u_int32 lost_pkt;       /* # of transport packets has
                           /* been lost */
   u_int32 op_status;      /* operation status */
} pid_status_blk *Pid_status_blk;
```

Description

`dxm_pid_status()` copies the status information from the specified PID into the status block.

Parameters

`dxmsp`	Points to DUXMan global storage
`pbp`	Points to the parameter block

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

Called By

_os_dxm_pid_status()

dxm_setstat()

DUXMan setstat Function

Syntax

```
#include <dxm_sys.h>
error_code dxm_setstat(
     dxm_stat     *dxmsp,
     dxm_gs_ss_pb *pbp);
```

Parameter Block

```
typedef struct _dxm_gs_ss_pb
{
   u_int16 scode;          /* function code */
   u_int16 prm_size;       /* parameter size */
   void *prm_blk;          /* pointer to parameter block */
} dxm_gs_ss_pb, *Dxm_gs_ss_pb;
```

Description

dxm_setstat() calls the hardware-specific setstat function hw_dxm_setstat() to process a specific DUXMan setstat call.

Parameters

dxmsp	Points to DUXMan global storage
pbp	Points to the parameter block

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

Called By

_os_dxm_setstat()

Hardware-Specific Layer Functions

DUXMan hardware-specific layer functions must be customized for a specific port. These functions control the specific TDIC used to implement the required functions. The following table summarizes the DUXMan hardware-layer functions. All of these functions are called by the common-layer code. To port DUXMan, these functions need to be re-written.

Table 3-3 DUXMan Hardware-Specific Layer Functions

Function	Description
`hw_dxm_flush()`	Disables DUXMan
`hw_dxm_flush_pat()`	Flush PAT Cache
`hw_dxm_getstat()`	DUXMan getstat Call
`hw_dxm_init()`	Initializes Hardware
`hw_dxm_setstat()`	DUXMan setstat Call
`hw_dxm_term()`	De-initializes Hardware
`hw_insert_event()`	Inserts an Event
`hw_pid_abtsec()`	Abort the dxm_pid_getsect Call
`hw_add_buf()`	Adds a Buffer to an Active PID Stream
`hw_pid_chgsec()`	Change the Mask and Value
`hw_pid_delete()`	Deletes a PID from the Active PID Table
`hw_pid_getpsi()`	Gets PSI Table
`hw_pid_getsec()`	Get PSI/SI Sections

Table 3-3 DUXMan Hardware-Specific Layer Functions (continued)

Function	Description
`hw_pid_insert()`	Inserts a PID into the Active PID Table
`hw_remove_event()`	Removes an Event

hw_dxm_flush()

Disables DUXMan

Syntax

```
#include <dxm_sys.h>
error_code hw_dxm_flush(dxm_stat *dxmsp);
```

Description

`hw_dxm_flush()` clears all the active PIDs and resets the hardware to its initial state.

Parameters

`dxmsp`	Points to DUXMan global storage

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dxm_flush()`

See Also

`hw_dxm_init()`
`hw_dxm_term()`

hw_dxm_flush_pat()

Flush PAT Cache

Syntax

```
#include <dxm_sys.h>
error_code hw_dxm_flush_pat (dxm_stat *dxmsp);
```

Description

Locate the entry for PAT and disable the version filtering.

Parameters

`dxmsp`	Points to DUXMan global storage

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dxm_flush_pat()`

See Also

`hw_dxm_flush()`

hw_dxm_getstat()

DUXMan getstat Call

Syntax

```
#include <dxm_sys.h>
error_code hw_dxm_getstat(
     dxm_stat     *dxmsp,
     dxm_gs_ss_pb *pb);
```

Description

`hw_dxm_getstat()` dispatches hardware-specific `getstat` calls to the appropriate functions. The functionalities supported are hardware dependent. Any functionality not supported in the common code can be specified here.

Parameters

`dxmsp`	Points to DUXMan global storage
`pb`	Points to the parameter block

Non-Fatal Errors

OS-9 error code or `SUCCESS`	`(0)` if no error occurred.
`EOS_UNKSVC`	Returned when the `getstat` code is unrecognized

Called By

`dxm_getstat()`

hw_dxm_init()

Initializes Hardware

Syntax

```
#include <dxm_sys.h>
error_code hw_dxm_init(dxm_stat *dxmsp);
```

Description

`hw_dxm_init()` initializes any hardware or variables specific to the hardware, such as resetting the TDIC, setting up the base addresses of the TDIC, downloading microcode (if any), and installing interrupt service routines.

Parameters

`dxmsp`	Points to DUXMan global storage

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dvm_init()`

hw_dxm_setstat()

DUXMan setstat Call

Syntax

```
#include <dxm_sys.h>
error_code hw_dxm_setstat(
     dxm_stat     *dxmsp,
     dxm_gs_ss_pb *pb);
```

Description

hw_dxm_setstat() dispatches hardware-specific setstat calls to the appropriate functions. The functionalities supported are hardware dependent. Any functionality not supported in the common code can be specified here.

Parameters

dxmsp	Points to DUXMan global storage
pb	Points to the parameter block

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

EOS_UNKSVC	Returned if setstat code is unrecognized

Called By

dxm_setstat()

hw_dxm_term()

De-initializes Hardware

Syntax

```
#include <dxm_sys.h>
error_code hw_dxm_term(dxm_stat *dxmsp);
```

Description

`hw_dxm_term()` de-initializes the TDIC and removes the installed interrupt service routine.

Parameters

`dxmsp`	Points to DUXMan global storage

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dvm_term()`

hw_insert_event()

Inserts an Event

Syntax

```
#include <dxm_sys.h>
error_code hw_insert_event(
     dxm_stat     *dxmsp,
     sib *csib, eqe *ev);
```

Description

`hw_insert_event()` sets up the appropriate interrupts and flags according to the event type.

Parameters

`dxmsp`	Points to DUXMan global storage
`csib`	Points to the current SIB entry
`ev`	Points to the current event entry

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dxm_pid_event()`

hw_pid_abtsec()

Abort the dxm_pid_getsect Call

Syntax

```
#include <dxm_sys.h>
error_code hw_pid_abtsect(
     dxm_stat     *dxmsp,
     sqe          *csqe);
```

Description

Locate the entry in the hardware table and clear the table ID register. If this is the last entry for this SIB, delete the PID pointers.

Parameters

`dxmsp`	Points to DUXMan global storage
`csqe`	Points to the current SQE

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dxm_pid_abtsect()`

See Also

`hw_pid_chgsec()`
`hw_pid_getsec()`

hw_add_buf()

Adds a Buffer to an Active PID Stream

Syntax

```
#include <dxm_sys.h>
error_code hw_add_buf(
     dxm_stat *dxmsp,
     sib *csib);
```

Description

hw_add_buf() associates an added buffer with the proper hardware output.

Parameters

dxmsp	Points to DUXMan global storage
csib	Points to the current SIB entry

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

Called By

dxm_pid_addbuf()

See Also

hw_pid_delete()
hw_pid_insert()

hw_pid_chgsec()

Change the Mask and Value

Syntax

```
#include <dxm_sys.h>
error_code hw_pid_chgsec(
     dxm_stat     *dxmsp,
     sqe          *csqe);
```

Description

Makes changes to the related registers based on the flags set in the common code.

Parameters

`dxmsp`	Points to DUXMan global storage
`csqe`	Points to the current SQE

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dxm_pid_chgsect()`

See Also

`hw_pid_abtsec()`
`hw_pid_getsec()`

hw_pid_delete()

Deletes a PID from the Active PID Table

Syntax

```
#include <dxm_sys.h>
error_code hw_pid_insert(
     dxm_stat      *dxmsp,
     sib           *csib);
```

Description

`hw_pid_delete()` disables the output of the specified PID stream in hardware.

Parameters

`dxmsp`	Points to DUXMan global storage
`csib`	Points to the current SIB entry

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dxm_pid_delete()`

See Also

`hw_pid_insert()`
`hw_pid_abtsec()`

hw_pid_getpsi()

Gets PSI Table

Syntax

```
#include <dxm_sys.h>
error_code hw_pid_getpsi(
     dxm_stat     *dxmsp,
     sib          *csib,
     tqe          *ctqe);
```

Description

`hw_pid_getpsi()` sets up hardware to parse the PSI sections, involving setting up the time-out timer, checking the table id and version number, and setting up proper registers accordingly. It returns when one of the following cases is met:

- The requested table is copied to the buffer

or

- It times out because no table has been found or copied

Parameters

`dxmsp`	Points to DUXMan global storage
`csib`	Points to the current SIB entry
`ctqe`	Points to the requested table entry

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dxm_pid_getpsi()`

hw_pid_getsec()

Get PSI/SI Sections

Syntax

```
#include <dxm_sys.h>
error_code hw_pid_getsec(
     dxm_stat     *dxmsp,
     sqe          *csqe);
```

Description

If this is the first table for this SIB, call hw_pid_insert() to insert the new PID. Set up the registers on the hardware and enable passing of the PID and sector data.

Parameters

dxmsp	Points to DUXMan global storage
csqe	Points to the current SQE

Non-Fatal Errors

OS-9 error code or SUCCESS (0) if no error occurred.

Called By

dxm_pid_getsect()

See Also

hw_pid_abtsec()
hw_pid_chgsec()

hw_pid_insert()

Inserts a PID into the Active PID Table

Syntax

```
#include <dxm_sys.h>
error_code hw_pid_insert(
     dxm_stat     *dxmsp,
     sib          *csib);
```

Description

`hw_pid_insert()` enables the specified PID stream to be output to its destination in hardware. This includes setting up the stream parsing mode and output data buffer for the hardware.

Parameters

`dxmsp`	Points to DUXMan global storage
`csib`	Points to current SIB entry

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dxm_pid_insert()`

See Also

`hw_pid_abtsec()`
`hw_pid_delete()`

hw_remove_event()

Removes an Event

Syntax

```
#include <dxm_sys.h>
error_code hw_remove_event(
     dxm_stat     *dxmsp,
     sib          *csib,
     eqe          *eqe);
```

Description

`hw_remove_event()` disables the previously activated interrupt and resets related flags according to the event type.

Parameters

`dxmsp`	Points to DUXMan global storage
`csib`	Points to the current SIB entry
`eqe`	Points to the current event entry

Non-Fatal Errors

OS-9 error code or `SUCCESS (0)` if no error occurred.

Called By

`dxm_pid_event()`

Making DUXMan Modules

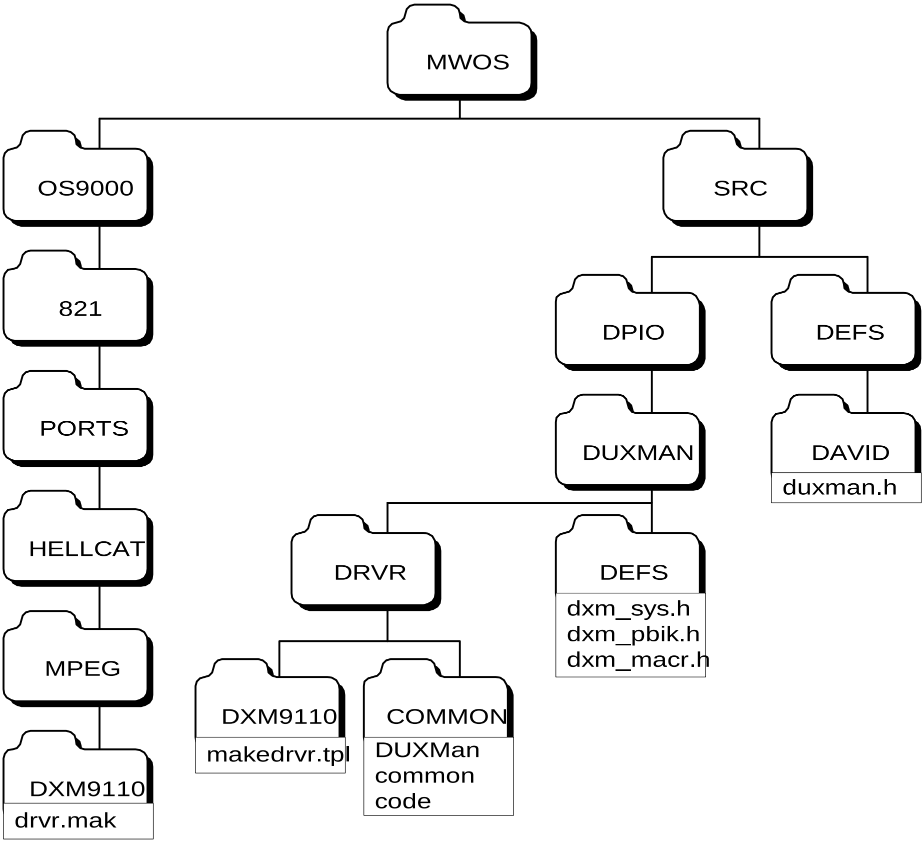
DUXMan has two object modules: the device driver/device manager module `dxmxxx` and the device descriptor module `dux`.

DUXMan is a dual-ported I/O (DPIO) product; this means it can be compiled for OS-9 systems.

To Make the Driver Module

The directory tree to make the DUXMan module is shown in **Figure 3-5**.

Figure 3-5 Directory Tree of DUXMan Module



The file `makedrvr.tpl` located in the `/DXM9110` directory is part of the makefile required to make the DUXMan driver module for the Hellcat.

The root makefile `drvr.mak` for the Hellcat port is located at:

`MWOS/OS9000/821/PORTS/HELLCAT/MPEG/DXM9110`

To make a DUXMan module for the DV340 port, enter the following commands:

```
$ cd /MWOS/OS9000/821/PORTS/HELLCAT/MPEG/DXM9110
$ os9make -uf drvr.mak
```

The newly created DUXMan module will be located at:

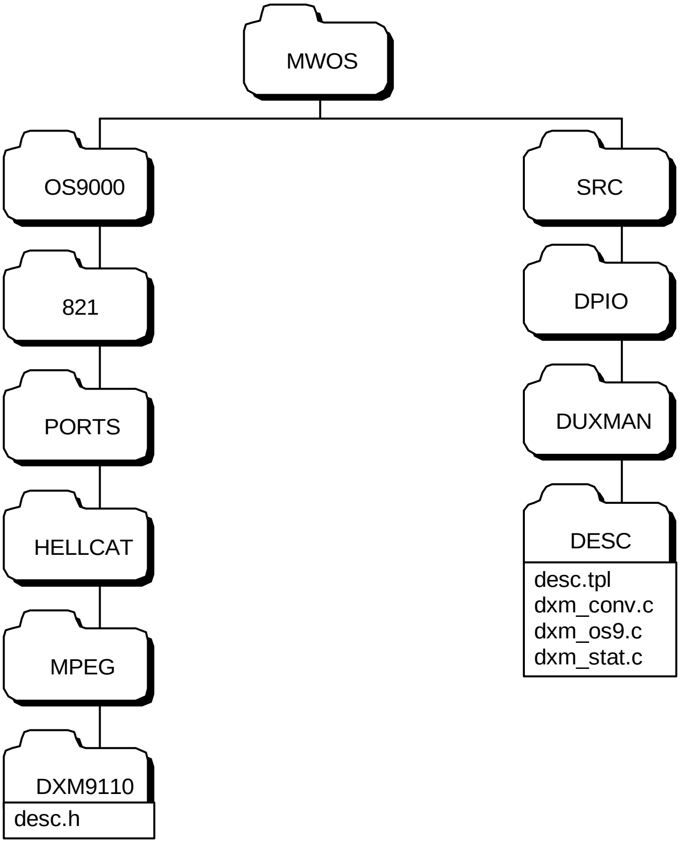
`/MWOS/OS9000/821/PORTS/HELLCAT/CMDS/BOOTOBJS/MPEG/dxm9110`

To Make the Device Descriptor Module

The device descriptor module stores the data to initialize the DUXMan static storage when it is initialized for the first time. All fields in the static storage, including hardware specific field `DRVR_HW_SPECIFIC` if it exists, need to be initialized. Four files are involved in creating the DUXMan device descriptor module. Three of them are C source code, located in the directory `$MWOS/SRC/DPIO/DUXMAN/DESC`. The other file is `desc.h`.

The locations of all necessary files are shown in **Figure 3-6**.

Figure 3-6 Directory Tree for DUXMan Device Descriptor



The file `desc.h` must be customized for each port. An example file is shown below.

```
/*************************************************************/
/* DUXMAN descriptor variable definitions for HELLCAT port   */
/*************************************************************/

#ifndef _DXM_DES_H
#define _DXM_DES_H

#include <types.h>
#include <cl9110.h>
#include <dxm_port.h>
#include <memory.h>

#ifndef VIDEO1
#define VIDEO1  0x80
#endif

#ifdef SHARABLE
#define DESCMODES_IREAD|S_IWRITE
#else
#define DESCMODES_IREAD|S_IWRITE|S_ISHARE
#endif

#define DXM_DRIVERNAME"dxm9110"
#define DXM_PORTADDR0x60000000
#define DXM_LUN0
#define DXM_VECTOR0x44
#define DXM_IRQLEVEL0
#define DXM_PRIORITY20
#define MAX_PID_ENTRIES16
#define MAX_EV_ENTRIES10
#define MAX_DEV_ENTRIES2
#define DXM_DEBUG_SIZE 1024*64/* v_debug_size */

#define PAT_BUF_SIZE1024/* PAT buffer size */
#define PRV_BUF_SIZE0/* private data buffer size */
#define PRV_BUF_MCOLOR MEM_ANY/* private buffer memory color */
#define PRV_BUF_COUNT8/* private buffer count */
#define DXM_SYSMEM_SIZE0/* demuxer special ram size */
#define MAX_TB_ENTRIES0x02/* max num of table entries */
      /* for each PID */
#define DXM_SYSFLAGS0
```

```
#define DEV_SPECIFIC_VALS
   NULL,/* cl9110 register map address */\
   NULL,/* cl9110 PID table start address */\
   NULL,/* cl9110 PTS table start address */\
   NULL,/* cl9110 segment parsing table */\
   "cl9110.ucd",/* v_ucode[12]: microcode module name */\
   0, \
   {0},\
   {0},/* v_qeueus[16]: cl9110 cache queue */\
   NULL,/* v_pat_prime: PAT primary buffer */\
   NULL,/* v_pat_second: PAT secondary buffer */\
   0,/* v_pat_rev:Revision # of PAT table */\
   0,/* v_pat_complete_flag:curr pat tab done */\
   0,/* v_reserv2: reserved fields */\
   0,/* v_pcr_delta_count:pcr delta position */\
   {0},/* v_pcr_delta_array:pcr delta array */\
   0,/* v_pcr_delta_average: averaged delta */\
   0,/* v_ev_creat_flag: OS9 event flag */\
   0,/* v_ev_id: OS9 event id */

#endif /* _MPEG */
```

Note

The definition for macro `DEV_SPECIFIC_VALS` should match the macro `DRVR_HW_SPECIFIC` defined in `dxm_port.h`.

To make a DUXMan device descriptor for the Hellcat, enter the following commands:

```
$ cd MWOS/OS9000/821/PORTS/HELLCAT/MPEG/DXM9110
$ os9make -uf desc.mak
```

The newly created DUXMan descriptor (named dux) is located in the following directory:

```
MWOS/OS9000/821/PORTS/HELLCAT/CMDS/BOOTOBJS/MPEG
```

Appendix A: Stream API

The Stream API allows you to asynchronously route data from the demultiplexer to either the MPEG decoder or to system memory.

This appendix contains the following sections:

- **Data Format**
- **Stream Control Blocks**
- **Stream Control Lists**
- **Stream Termination**
- **Buffer Full**
- **Stream API Programming Reference**

Data Format

The network delivers MPEG-2 Transport Streams (MPTS) to DUXMan (Stream Demultiplexer). Each MPTS contains one or more programs, and each program contains one or more multiplexed elementary streams/private data streams.

Elementary streams contain MPEG Audio/Video (A/V) data carried in Packetized Elementary Stream (PES) packets. Non-MPEG data is carried in private data streams.

Both the PES and private data streams are inserted into 188-byte MPEG-2 transport packets. Each packet has a 4-byte header containing a 13-bit Packet Identification (PID). The PID identifies, via the Program Specific Information (PSI) tables, the contents of the data in the transport packet. Transport packets of one PID value carry data of only one stream. Thus, the PID uniquely identifies the stream and is used by DUXMan for demultiplexing. DUXMan allows any stream to be directed into system RAM or sent directly to the MPEG A/V subsystem.

For More Information

For more information about MPEG-2 and MPTS, refer to ***ISO/IEC 13818-1:1996 Information Technology — Generic Coding of Moving Pictures and Associated Audio Information Systems***

MPEG-2 Transport Packet Format

The transport packet format contains:

- A 4-byte header
- Optional adaptation field
- Data

Table A-1 Transport Header

Field	# of Bits
sync_byte	8
transport_error_indicator	1
payload_unit_indicator	1
transport_priority	1
PID	13
transport_scrambling_control	2
adaptation_field_control	2
continuity_counter	4

Relevant transport packet header fields are described below:

sync_byte	A fixed 8-bit field with a value of $47
payload_unit_indicator	A 1-bit flag which has a normative meaning for transport packets carrying PES packets or PSI data

When the transport packet's payload contains PES data, the payload_unit_indicator has the following significance:

1 =	Payload of this transport packet begins with the first byte of a PES packet
0 =	No PES packet will start in this transport packet

When the transport packet's payload contains PSI data, the `payload_unit_indicator` has the following significance:

1 =	The first byte of this transport packet's payload carries the `pointer_field`
0 =	The first byte of this transport packet's payload does not carry the `pointer_field`*.* There is no `pointer_field` in the payload.

For private data packets, the meaning of this bit is defined as "not defined".

`PID` is a 13-bit field indicating the type of data stored in the packet payload.

Table A-2 PID Values

PID Value	Reserved For
`0x0000`	Program Association Table (PAT)
`0x0001`	Conditional Access Table (CAT)
`0x0002 - 0x000F`	Reserved
`0x1FFF`	Null packets

`adaptation_field_control` is a 2-bit field indicating whether this transport packet header is followed by an adaptation field and/or payload. MPEG-2 decoders discard transport packets with a value of `00` in `adaptation_field_control`.

Table A-3 `adaptation_field_control` Values

Value	Description
`00`	Reserved by ISO
`01`	No adaptation field, payload only

Table A-3 `adaptation_field_control` Values

Value	Description
`10`	Adaptation field only, no payload
`11`	Adaptation field followed by payload

Stream Control Blocks

DAVID applications use Stream Control Blocks (SCB) to control data flow from DUXMan. The SCB identifies the PID required by the application and provides destination buffers for data in the packet identified by that PID. Also, the SCB defines how much of the packet the application requires.

An MPTS packet always has a 4-byte header. The header is followed by an adaptation field, payload data, or both. Flags in the SCB direct DUXMan to operate in one of the following modes for the specified PID:

- Hardware Direct
- Private Data: RAW or PAYLOAD only

In Private Data RAW mode, the entire 188-byte packet is copied into the destination buffers. In Private Data PAYLOAD mode, only the payload portion of the transport packet is copied into the destination buffers. If a packet contains an adaptation field, it is treated as padding and the remaining bytes of data in the transport packet are treated as payload data.

In hardware direct mode, data is transferred directly by the hardware from the network to the MPEG decoders.

SCB Format

```
typedef struct _scb {
   u_int16 scb_stat;/* Current status of the play */
   u_int16 scb_sig;/* Signal to be sent on */
      /* termination */
   u_int 16 scb_pid;/* Identifier for stream */
   U_int 16 scb_mode;/* Stream Mode */
   u_int32 scb_err;/* Error Code */
   scl *scb_scls;/* pointer to a linked list of */
      /* scl structures */
   u_int16 scb_in_stream_type;
   u_int16 scb_out_stream_type;
   u_int16 scb_version;
   u_int32 scb_transaction_id;/* transaction ID field for */
      /* NETWORK */
   u_char scb_rsvd[12];/* Reserved field, be sure to */
```

```
      /* set to 0 */
} scb, *Scb;
```

The SCB parameters are defined as follows:

scb_stat

The current status of the stream operation. scb_stat should initially be set to zero by the application. It is updated by the system at the time an SCL is filled. The application may read scb_stat at any time during the stream operation to determine its current status, but should not modify it. The bits in scb_stat are defined as follows:

Table A-4

Bit Number	Description/Defines
0-7	Reserved (must be zero)
8	Stream done (SCB_S_DN)
9-14	Reserved (must be zero)
15	Error Condition (SCB_S_ER)

scb_sig

The signal number to send to the application when the stream operation is terminated, or an error occurs. The application may change scb_sig any time during the stream operation. If scb_sig is set to zero, no signal is sent when the stream is finished. When the application receives this signal, it can check the scb_stat field for errors or if the stream has finished.

`scb_pid` The packet ID of the incoming MPEG-2 data. Only packets labeled with this PID are channeled to the SCL buffers or hardware associated with this SCB.

`scb_mode` A bit mask that determines the mode of the stream call.

Table A-5 `scb_mode` Bit Mask

Bit Number	Description
0-3	Reserved (must be zero)
4-6	000 = Reserved 001 = Private Data mode (`SCB_M_SCL_DIRECT`) 010 = PCR mode (`SCB_M_NULL_OUT`) 100 = Hardware Direct mode (`SCB_M_HW_DIRECT`)
7-15	Reserved (must be zero)

`scb_err` When an error occurs in a stream operation, the appropriate error code is inserted into `scb_err` and the signal in `scb_sig` is sent to the application. It is the application’s responsibility to check and clear this field.

`scb_scls` This field points to the next SCL to write data associated with the PID identified in `scb_pid`. A NULL `scb_scls` pointer indicates that, if possible, any data associated with the `scb_pid` should be sent directly to the MPEG processor without placing it in memory. An error is returned if this is not possible.

This field is initialized before the stream is started by the application and is updated by the system to point to the next SCL in the list as each SCL is filled. This field may be changed by the application.

`scb_in_stream_type`

A bit mask that determines the type of input stream on the PID value specified in `scb_pid`.

When using an `scb_mode` of `SCB_M_HW_DIRECT`, `scb_in_stream_type` can use one of the following values:

```
MPEG_1_VIDEO
MPEG_1_AUDIO
MPEG_2_VIDEO
MPEG_2_AUDIO
```

The following flag can be used in combination with `MPEG_2_VIDEO`:

`PCR_PID_FLAG` in combination with `MPEG_2_VIDEO` indicates that the PCR data for the video is also on this PID.

When using an `scb_mode` of `SCB_M_NULL_OUT`, `scb_in_stream_type` should be set to `PCR_PID_FLAG` indicating that the data on the specified PID is the PCR data for the video PID.

When using an `scb_mode` of `SCB_M_SCL_DIRECT`, `scb_in_stream_type` can use one of the following values:

```
STREAM_TYPE_UNKNOWN
PVT_13818_1
```

scb_out_stream_type

Unless scb_mode is set to SCB_M_SCL_DIRECT, this must be set to STREAM_TYPE_UNKNOWN.

When using SCB_M_SCL_DIRECT mode, scb_out_stream_type can be one of the following:

PVT_13818_1
PVT_PES

The following flags may also be used with the above values:

SCB_RAW_FLAG Causes the data to be transferred to your SCL buffer(s) without removing the transport layer

SCB_PAYLOAD_FLAG
Causes the data to be transferred to your SCL buffer(s) after removing the transport layer and any adaptation fields.

scb_version

Version of the SCB typedef. Should be set to VERSION2_0.

scb_transaction_id

Currently unused. Should be 0.

scb_rsvd

Reserved. Should be 0.

Stream Control Lists

The SCL refers to a Stream Control List (SCL). The SCL is a linked list of structures that point to buffers for the packet data.

SCL Format

```
typedef struct _scl
{
   u_char  scl_ctrl;       /* Control byte */
   u_char  scl_pheader[3]; /* Currently unused */
   u_int16 scl_sig;        /* Signal to be sent on buffer full */
   struct  _scl
           *scl_nxt;       /* Pointer to next scl */
   u_char  *scl_buf;       /* Pointer to buffer */
   u_int32 scl_bufsz;      /* Size of buffer */
   u_int16 scl_err_sig;    /* Currently unused */
   u_char  scl_rsv[4];     /* reserved, be sure to set to 0 */
   u_int32 scl_cnt;        /* Current offset in buffer */
} scl, *Scl;
```

The SCL parameters are defined as follows:

`scl_ctrl`

The current status of the buffer. `scl_ctrl` should be initialized to zero by the application before the stream operation is started.

This field will be set to `SCL_C_BFULL` when the SCL buffer is full.

`scl_pheader`

Currently not used. The application should initialize this field to zeros.

`scl_sig`

The signal number to send to the application when the SCL buffer is full. The application may change `scl_sig` at any time during stream operation execution. If `scl_sig` contains zero, no signal is sent.

`scl_nxt`

This is the pointer to the next SCL element. The linked list of SCLs, which is built by the application, contains at least one element. The list may be circular or may be terminated by the entry of a zero in this field. The list of SCLs may be manipulated by the application at any time during the stream operation.

When the buffer for an SCL is full, the contents of this field are placed in the `scb_scls` entry of the SCB that references this SCL.

`scl_buf`

The address of the buffer in which to store the data from the network. The buffer must start at a word boundary. If this pointer is NULL, the stream call is aborted. The application initializes `scl_buf` and may change it at any time during the stream operation.

`scl_bufsz`

Size in bytes of the buffer pointed to by `scl_buf`. This field is initialized by the application and may be changed by the application at any time during the execution of the stream operation. The buffer must be a multiple of 188 bytes in RAW mode and 184 bytes in PAYLOAD mode.

Note

The buffer size should be as large as possible to minimize the processing involved with every SCL.

`scl_err_sig`

Not currently used. Should be 0.

`scl_rsv[6]`

Reserved for compatibility with CDFM. The application should initialize this field to zeros.

`scl_cnt`

The offset in bytes into `scl_buf` of the next position to copy data. It should be initialized to zero by the application before the start of the stream operation. This field is updated by the system, but it may be changed by the application at any time during the stream operation.

Stream Termination

Any of the following conditions can cause the stream operation to terminate:

- The end of a null terminated SCL list is reached. Bit 8 (`SCL_C_BFULL`) is set in the `scb_stat`.
- For a circular list, if the buffer full bit of the current SCL is still set when the driver attempts to fill the buffer (this implies that all buffers in the list are full), the stream operation is aborted. Bit 8 (`SCL_C_BFULL`) is set in `scb_stat` and `E_DEVOVF` is returned in the `scb_err` field of the SCB.
- A software abort is received (`_os_ss_abortstream()`). Bit 8 (`SCB_S_DN`) and bit 15 (`SCB_S_ER`) are set in `scb_stat` and `E_ABORT` is returned in the `scb_err` field of the SCB.
- A hardware or software fatal error occurs (for example, network connection lost, time-out). The appropriate error code is inserted into the `scb_err` field and `SCB_S_ER` is set in `scb_stat`.

Buffer Full

The following conditions can cause a buffer full condition:

`scl_cnt` equals `scl_bufsz`.

The following steps are required to reset an SCL:

```
scl->scl_ctrl = 0;
scl->scl_cnt = 0;
```

Stream API Programming Reference

This section documents the Stream API system call interface and describes each Stream API system call.

Stream API Function Calls

Table A-6 lists and briefly describes the Stream API function calls. Detailed descriptions follow.

Table A-6 Stream API Library Functions

Function	Description
`_os_ss_abortstream()`	Abort Asynchronous Read
`_os_ss_readstream()`	Begin Asynchronous Read

_os_ss_abortstream()

Abort Asynchronous Read

Syntax

```
#include <DAVID/stream.h>
error_code _os_ss_abortstream (
     path_id      path,
     scb          *scb);
```

Description

_os_ss_abortstream() aborts the active asynchronous operation associated with the scb. If an asynchronous _os_ss_readstream() request is in progress, the signal in the scb_sig parameter is sent, the SCB_S_DN and SCB_S_ER bits are set in scb_stat, and an E_ABORT error code is copied into the scb_err field. If this scb is not active within an _os_ss_readstream(), the call returns E_NOPLAY. E_ILLPRM is returned if scb is NULL.

Errors

E_BPNUM	The path number specified in path is not valid
E_NOPLAY	The PID specified in scb is not being used in an asynchronous read operation
E_ILLPRM	The value of scb is NULL
E_IPRCID	The asynchronous read operation was not started by this process

_os_ss_readstream()

Begin Asynchronous Read

Syntax

```
#include <DAVID/stream.h>
error_code _os_ss_readstream (
     path_id      path,
     scb          *scb)
```

Description

_os_ss_readstream() asynchronously reads data from the network for a specific PID. The operation is controlled by a Stream Control Block (SCB) and a linked list of Stream Control List (SCL) structures that are allocated and initialized by the application. A pointer to the SCB is passed as a parameter to _os_ss_readstream().

Each SCL points to a data buffer. When a data packet comes in over the network, it is inserted into the current SCL's data buffer. When the SCL's buffer is filled, the buffer full bit of the SCL is set and the SCL signal is sent to the process that called _os_ss_readstream().

The SCL list can be either curricular or null terminated. At the end of null-terminated lists, the signal in the scb_sig field is sent to the application; and the SCB_S_DN bit is set in the scb_stat field to indicate the end of the stream operation. Data received after that is lost.

If an _os_ss_readstream() call is already active on this PID, E_DEVBSY is returned. E_FULL is returned if the maximum number of read stream calls is exceeded. This value is stored in the descriptor and can be changed. Its default is 16. E_BMODE is returned if the SCB_M_HW_DIRECT mode is set in the SCB and the hardware cannot support the mode.

Errors

E_BPNUM	The path number specified in path is not valid
E_DEVBSY	The PID specified in scb is already in use by another asynchronous read operation

`E_ILLPRM`	The value of `scb` is NULL
`E_FULL`	The maximum number of concurrent asynchronous read operations has been reached
`E_BMODE`	The SCB specifies a mode of `SCB_M_HW_DIRECT` and the hardware does not support it

Index

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