
# DAVID® Utilities and Applications

## Version 2.5

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 2.5 of DAVID.
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.

November 2001

# Table of Contents

# Chapter 1: Using moplay

This chapter describes the `moplay` example program which is used as an example of how to:

- Play MPEG with MPFM.
- Request MPEG streams using UpLink.
- Using a MAUI drawmap to decode MPEG into.

The source code for `moplay` is located in $MWOS/SRC/DAVID/DEMOS.

# moplay Overview

This section gives an overview of `moplay`, its command line parameters and usage.

`moplay` was designed to provide an example of how to play MPEG streams arriving over a network.

## moplay Command Line Options

If no PIDs are specified (or only one) using the command line options below, the PAT and PMT will be parsed from the incoming stream to determine which PIDs to use. This is accomplished using the DAVID Demo Library function `david_demo_get_pids()`.

### -apid=<audio PID(0xNNNN)>

Allows the specification of a PID value to decode as the audio portion of the MPEG stream.

### -vpid=<video PID(0xNNNN)>

Allows the specification of a PID value to decode as the video portion of the MPEG stream.

### -ppid=<PCR PID(0xNNNN)>

If the PCR data for the stream is not in the video PID stream, this options is used to specify the PID the PCR data will arrive on.

### -ac3

If the audio PID stream carries Dolby AC-3 audio, this option is used to specify that. The default is standard MPEG-2 audio.

### -u=<path to mpeg stream>

If `moplay` was compiled to use UpLink, this option is used to specify a pathlist to the location of the MPEG stream to send over the network to the device.

If this option is not specified, `moplay` assumes that the MPEG stream is already being delivered over the network.

## moplay Usage

While `moplay` is playing a stream, the following keystrokes cause the following actions:

- 'q' to Quit
- 'p' to Pause
- 'c' to Continue
- '?' to receive this message

# Chapter 2: Using the systrap Handler

This chapter describes the `systrap` module verification API. The functions are contained in a system state trap handler. The functions allocate memory for modules, and link and unlink modules.

# Introduction to systrap

This chapter describes the `systrap` module verification API. The functions allocate memory for modules, link and unlink modules, and are contained in a system state trap handler.

## Function Reference

This section provides detailed information about the functions in this API. These functions are the complete and only interface to this trap handler.

**Table 2-1 Function Calls in systrap Handler**

| Functions | Description |
|---|---|
| `systrap_init()` | Link to the Trap Handler |
| `systrap_alloc()` | Allocate Memory in System State |
| `systrap_link()` | Verify Binary Large OBject (BLOB) of Memory as Module(s) |
| `systrap_unlink()` | Unlink Module(s) |
| `systrap_free()` | Free Memory Allocated by systrap_alloc() |
| `systrap_ov_free()` | Free Safety Areas |

When using this API, include `st_lib.h` in your source files and link with `st_lib.l`.

## systrap_init()

Link to the Trap Handler

### Syntax

```
systrap_init(void);
```

### Description

The function `_os_tlink()` is an application to the systrap trap handler. If the call to `_os_tlink()` fails, the `error_code` is returned to the caller. If successful, the function returns `0`. The application must be linked to `st_lib.l` to call this function.

### Parameters

None

# systrap_alloc()

Allocate Memory in System State

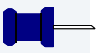
## Syntax

```
systrap_alloc(
     u_int32      *size,
     void         **mem_ptr);
```

## Description

This function allocates the specified amount of memory using `_os_srqmem()`. The caller should retain the value returned in size if `systrap_free()` will be called.

A safety area of 16 bytes on either side of the allocation is reserved. The total size of the allocation is stored in the first four bytes of the safety area. This allows the caller to discard the size of the allocation (see `systrap_free()`).



### Note

The 16-byte safety area is determined by OS-9 (which manages memory in 16-byte blocks.) A pointer to the memory immediately after the leading safety area is returned in `mem_ptr` and the total size of the allocation is returned in size. If the call to `_os_srqmem()` results in an error, this `error_code` is returned to the caller.

## Parameters

`size`

`mem_ptr`

## systrap_link()

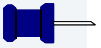
Verify Binary Large OBject (BLOB) of Memory as Module(s)

### Syntax

```
systrap_link(
     u_int32      grp_sz,
     mh_com       *mod_header,
     mod_dir      **entry);
```

### Description

This function validates a piece of memory as a module or modules. The buffer pointed to by `mod_header` is assumed to have been allocated by `systrap_alloc()`. This function increments the link count of the first module found in the buffer to 1 with `_os_link()`. When more than one module is validated, each module's module directory entry is updated with the group size and group pointer (`mod_header`), making the modules in the buffer a module group.



### Note

When successful, the safety area on both sides of the buffer is freed. After the buffer contains validated modules, the memory should not be deallocated. The deallocation takes place when the module(s) are removed from the module directory. If one or more calls to `_os9_vmodul()` fail, the `error_code` from the last failure is returned to the caller after the appropriate modules have been verified.

### Parameters

`grp_sz`

`mod_header`

`entry`

# systrap_unlink()

Unlink Module(s)

## Syntax

```
systrap_unlink(mh_com *mod_header)
```

## Description

This function is a wrapper around `_os_unlink()` included for completeness.

## Parameters

`mod_header`

## systrap_free()

Free Memory Allocated by systrap_alloc()

### Syntax

```
systrap_free(void *mem_ptr);
```

### Description

This function deallocates memory allocated by `systrap_alloc()`. This function calls `_os_srtmem()` with the given size and memory pointer. It returns the status of the call to `_os_srtmem()`. It uses the total size of the allocation stored in the first four bytes of the leading safety area and the value of `mem_ptr` to call `_os_srtmem()`, which frees the buffer—including both safety areas. This function returns the `error_code` returned by the call to `_os_srtmem()`.

### Parameters

`mem_ptr`

# systrap_ov_free()

## Free Safety Areas

### Syntax

```
systrap_ov_free(void *mem_ptr);
```

### Description

This function is similar to `systrap_free()`. However, it only frees the safety area on either side of the allocation. It should not be called unless the rest of the buffer will never be deallocated. This function returns the `error_code` returned by the call to `_os_srtmem()`.

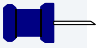
### Parameters

`mem_ptr`

# Chapter 3: Using Player Shell

The player shell is generally the first process to execute in a DAVID system. It is supplied with the operating system software to perform basic system initialization, channel tuning, and retrieve interactive applications from the network.

This chapter describes how to use the player shell and how to port it to your environment.



**Note**

The player shell provided with the DAVID Installation Pak illustrates one possible implementation of a player shell. You will want to port and implement your own version of the player shell for your specific network/STB deployment.

# Player Shell Overview

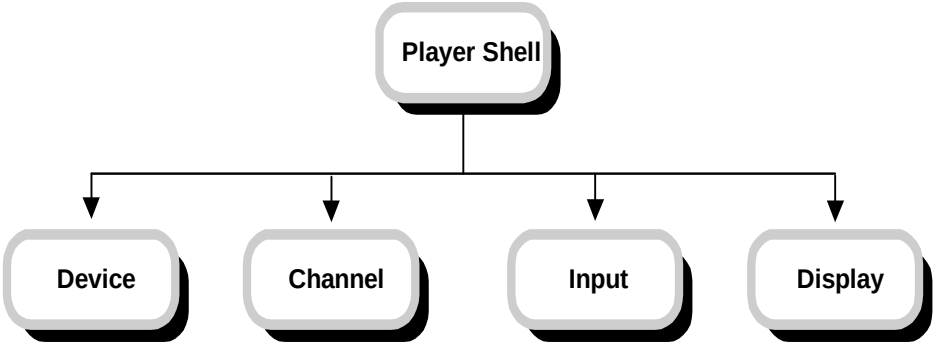
This section gives a brief overview of a fully functional player shell architecture, responsibilities, and requirements. This is provided as one possible OEM implementation.

Following this section is an overview of the architecture, responsibilities, and requirements of the sample player shell provided with both the DAVID Installation Pak and the DAVID Application Development Pak.

## Player Shell Architecture

The following figure illustrates the functionality among the various components comprising the player shell.

**Figure 3-1 Player Shell Architecture**




The device component performs all device initialization.

The Input component performs all input initialization. This is built over the Input API of MAUI. Typically the input process (`maui_inp`), upon which input processing depends, is launched at system startup when the player shell is launched.

The Channel component performs all channel ring management and channel tuning. It uses the Navigation Manager API to accomplish this.

### For More Information

Contact RadiSys Microware Communications Software Division, Inc. for information on how to acquire the Navigation Manager API.

The Display component performs display management using MAUI to display text and graphics.

## Player Shell Responsibilities and Considerations

### Common Player Shell Responsibilities

The player shell responsibilities common to all player shell implementations include:

- Initialize devices (which reduces memory fragmentation)
- Initialize MAUI APIs that are used by the player shell
- Start MAUI input process (if it was not already launched at system startup)
- Reserve remote keys that are exclusive to the player shell interface such as channel up, channel down, power on, and power off
- Handle default channel ring tuning such as channel up, channel down, last channel, and direct channel number entry

## Network-Dependent Responsibilities

The player shell network-dependent responsibilities include:

- Manage channel rings such as VIP rings, favorite rings, and parental control
- Handle pay-per-view (PPV) and subscriptions to premium channels
- Download and execute applications
- Establish communications with network provider (in an interactive network)

## Download and Execute Application Considerations

- Use the appropriate download protocol for the respective deployment (such as UpLink/`systrap`)
- Signal applications when events such as Emergency Broadcast System (EBS) messages occur or a user presses keys that cause applications to terminate (Power Off, Video Dial Tone (VDT), Video Information Provider (VIP))
- Devices pre-initialized
- Memory usage

## Other Player Shell Considerations

- Memory usage
- Adapting to graphics display characteristics
- Module naming conventions
- Modules that the player shell relies upon must be loaded into the module directory prior to executing the player shell

# Player Shell Functions

Player shell functions vary depending on the network architecture. As mentioned previously, the player shell is responsible for:

- Initializing devices
- Performing channel tuning
- Initiating communications with the network providers
- Downloading and executing applications

## Attaching Devices

When a device is attached under OS-9, the kernel calls the `attach` routines in the associated file manager which calls the `init` routine in the associated device driver. These routines are responsible for allocating file manager and driver memory used for storing global data. They are also responsible for initializing the hardware associated with the device and installing interrupt service routines (ISRs).

There are two reasons for the player shell to attach all of the primarily used devices as its first activity.

- The first is to force all memory allocations to be performed immediately and consecutively. This reduces problems with fragmentation if the device is initialized only when the applications open the devices.
- The second reason device initialization is done immediately is to prepare the system for the applications. With time-consuming activities — such as hardware initialization — completed, devices are ready to perform when applications need them.

## Establishing Network Communications

This task varies from network to network, as well as from interactive to broadcast environments. Initiating communications with a cable television (CATV) head-end system is much different than an ATM or Asymmetric Digital Subscriber Line (ADSL) network. This leads to one of the following scenarios.

- In the first, a network operator defines a set-top box (STB) that works in one, and only one, environment. The logic for this task may be built directly into the player shell and written into the system ROM.
- In the second case, the network environment is not defined and the player shell requires a protocol module to be downloaded from an external Network Interface Module (NIM) or from the network itself during the boot process. This protocol module contains the details of the network environment, since it is provided by the network operator.
- In the third case, the communication link may be on demand through a device such as a modem. For example, in a broadcast scenario it may be necessary to have the device's modem make a request for pay-per-view.

In an interactive network, this initial dialog with the network provides the user with a list of service providers and allows him or her to select one. Although the initial work in this area specified a generic command-oriented interface that was portable across any type of STB, it has the disadvantage of being less user-friendly. Newer directions point to the ability to download a special network application that allows for a richer graphics user interface.

## Downloading Applications (Interactive Television)

After you select a service provider, the player shell typically downloads an application to the STB. This application is delivered as one or more OS-9 modules. The player shell checks the cyclical redundancy check (CRC) on each module and, if intact, inserts them into the module directory. At this point, the player shell executes the first module and goes into a background mode.

The goal of the player shell is to prepare the system such that every time an application is downloaded and executed, it has a predictable environment, with a guaranteed number of system resources available. These system resources include:

- Free system RAM
- Free graphics RAM
- Uncontended CPU cycles
- Access to the full bandwidth of the network and audio/video processors

Having these resources available is the key to application portability. Generally, RAM is freely available in any consumer environment. Very few programs consume all 4-8 MB available on a typical personal computer. Programs that do require more system resources can ask the user to quit other currently executing applications. Otherwise, it is advisable for applications known to make heavy demands upon the STB to adapt to limited resources.

The idea that a program can ask a user for more space is not translatable to an STB. RAM is generally quite limited, and it is likely that every application uses all available RAM for buffering and downloading. Therefore, the player shell tries to guarantee that the application has what it needs by only allowing one application to be active in the STB at any given time.

## Optional Player Shell Features

The player shell can have many other responsibilities in addition to those mentioned above. Some of these depend on the network architecture; others depend only on the preferences of the network operator.

Architecture considerations are mainly dependent on the concept of broadcast networks. In a broadcast environment, the player shell is typically responsible for managing selection of the numerous analog and digital broadcast programs. This includes “surfing” (using the channel up and down buttons) as well as providing for favorite channel selections, using a program guide, or other preset mechanisms.

In most cases, the player shell is responsible for surfing, but program guides are often done by a special application either downloaded from the network or embedded in FLASH.

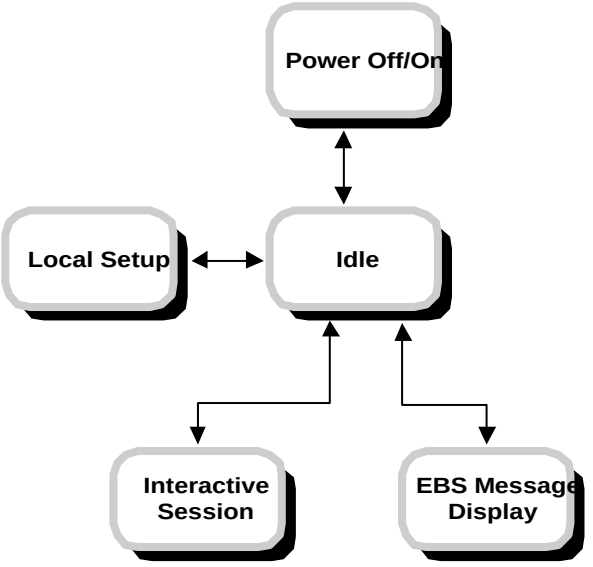
# Player Shell Operational Modes

## Operational Mode Descriptions

The player shell implements the following operational modes:

- Power off/on (standby)
- Idle/broadcast video
- Local setup
- Interactive session
- Emergency Broadcast System (EBS) or network system message display

**Figure 3-2 Player Shell Operation Modes**



## Power On/Off Mode

When the STB is in power off mode, only the remote control receiver is active. If the STB operates in a standby mode, the player shell uses this time to perform warm reboots of the system, for both security and performance reasons.

## Idle/Broadcast Mode

When the STB is first turned on, it is in an idle mode. In this mode, it routes any analog A/V signal received from an antenna, VCR, or standard CATV system directly through the STB.

If the STB is a hybrid analog/digital, or strictly digital system, the player shell allows the user to surf through the digital broadcast channels as well. If the channels are digital MPEG streams, the player shell routes them to the MPEG.

## Local Setup Mode

If the user presses the remote's Menu button, while the player shell is in **idle** mode, the player shell switches to local menu mode to display a menu of choices. This menu is manufacturer dependent. Possible menu options include:

- Select and store a preferred Video Information Provider (VIP)
- Set the STB identification (ID) number used to distinguish it from other STBs in the household
- Display program guides
- Set favorite channel selection maps

Selecting the Menu button a second time returns the STB to its **idle** mode. Local setup mode will likely not be available during interactive services.

### Interactive Session

If the user chooses to connect to the network, the player shell establishes the connection. After the selection is made, the player shell returns to **idle** mode.

### EBS Message Display

It is possible for the network provider to send EBS or other system messages to the STB. When the player shell detects these messages, it displays them using the STBs overlay graphics capabilities. Because the user may be involved in a highly interactive application, it may be necessary for applications to temporarily suspend operation while the message displays. Movies or other essentially linear applications may choose not to suspend their operations.

To display the EBS message, the player shell overlays the bottom 60 lines of the display. The bottom 20 lines are assumed to be non-visible; therefore, the text is centered in the upper 40 lines of the area. The text scrolls horizontally on the display as is typically seen in broadcast TV.

## Memory Use

The primary use of memory by the player shell is for graphics display. Since the player shell may be required to display a graphics overlay at any time, it permanently allocates a frame buffer from the video memory for that purpose. This frame buffer is 320x200 pixels. Other video data structures may also be maintained to aid in displaying this image.

If a multiplane graphics display is in use, this allocation comes from the front plane (Plane 0).

## Graphics Display Characteristics

The player shell respects the display safety area. In other words, it will not display any data in the display's top 20 or bottom 20 lines, or the left 20 or right 20 pixels.

All text is typically white on a blue background.

## Subscriptions and Pay per View

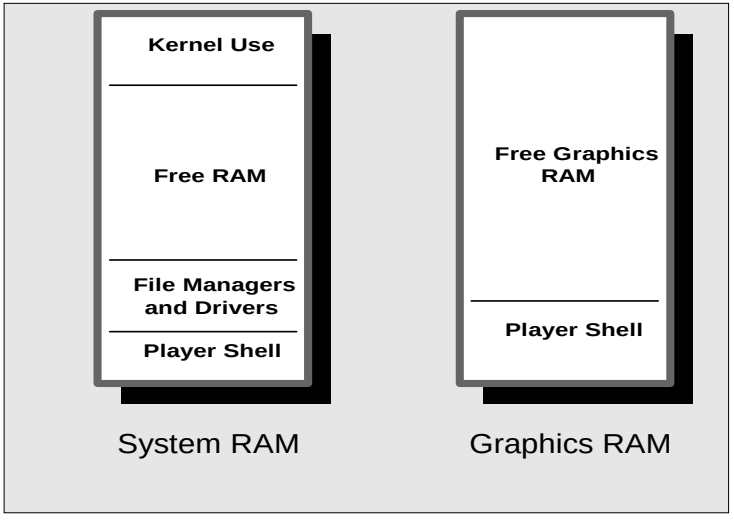
Pay-per-view (PPV) is accessible via an alternate (“barker”) channel and a PPV key on remote. The alternate barker channel appears when you attempt to change to a PPV channel. The STB displays a barker screen that enables you to request the channel via the PPV key. The barker channel displays instructions on which channel to tune to begin viewing the PPV channel. As a result of this activity, the VIP can bill the user for the service.

Premium channels can use the same technique as PPV channels. PPV channels are short-term subscriptions, while premiums are long-term.

## Application Startup Environment

The application in a STB environment must have a guaranteed set of system resources available to function properly. This applies primarily to system and graphics RAM, which might be used by the operating system or various background processes running in the system.

**Figure 3-3 Typical Memory Map**



The kernel uses a small amount of system memory at the lowest address available for its system data structures. This area includes the interrupt vectors, module directory, path table, and process table. Because the first application executed is the player shell, its data area is allocated from the highest addresses of free system RAM. The actual amount of RAM allocated depends on the number of features in the player shell.

An additional amount of RAM is used by the file managers and drivers for their data structures. The player shell initializes the system devices at start-up. This causes all of their global storage allocations to be concentrated together.

The result is a large, contiguous block of free memory available for applications. Each time an application finishes executing, the kernel frees up the memory used by it, preparing the system for the next application.

From this point forward, the kernel uses a very simple algorithm for memory allocations. When modules are downloaded to the system, they are placed at low addresses, moving higher. When memory is allocated by applications, memory is taken from high addresses, moving lower.

### For More Information

For a more complete discussion of memory management, refer to the OS-9 technical manuals.

## Application Naming Conventions

All modules in OS-9 have a name. When modules are loaded, they are placed in a **module directory**. To prevent name clashes in this directory, we highly recommend that modules included in ROM (including the entire DAVID OS) do NOT use the prefix `dvd_`.

On the other hand, any application modules downloaded to the system should start with the prefix `dvd_`.

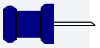
## Application Signaling

If the player shell receives an EBS message, it broadcasts a signal to applications that it downloaded and forked. This allows applications to temporarily suspend their activities while the message is displayed by the player shell.

Additionally, some buttons (Power) cause the current application to abort. Again, the player shell signals the application to inform it of the impending abort to allow it to save its status before exiting.

Three signals are reserved for this:

- `SIG_TERMINATE  0x18`
- `SIG_SUSPEND  0x19`
- `SIG_RESUME  0x87`



**Note**

Because signals SIG_SUSPEND and SIG_TERMINATE are below 0x20, they are considered fatal signals. That is, they terminate any blocking I/O operations when received. They do not affect the operation of asynchronous calls.

## SIG_TERMINATE

When the user requests the termination of the current application, the `SIG_TERMINATE` signal is broadcast to certain processes in the system. This signals an application that has one second to clean up and send any final messages to the server before being terminated by the player shell.

**Note**

When the user presses a button such as Power while an application is running, it is possible for the player shell to present the user with a prompt asking the user to verify the action (typically by pressing the button again). In this case, the player shell first broadcasts SIG_SUSPEND and then, if the action was verified, broadcasts SIG_TERMINATE.

### SIG_SUSPEND

The `SIG_SUSPEND` signal is broadcast to all processes in the system one second before the player shell needs to use the graphics display. This allows applications time to halt any animations or to send a message to the network telling it to pause any incoming data streams. It is not required that all applications immediately cease, but they should be aware some or all of the graphics overlay is taken over for the display of a system message.

### SIG_RESUME

The `SIG_RESUME` signal is broadcast to certain processes in the system immediately after the player shell has restored the previous graphics environment. The player shell will not destroy any graphics contexts in processes using the DAVID/MAUI interface to the graphics overlay functions, but applications may have to refresh (update) the display.

## Basic Channel Selection

After initialization is finished, the STB enters Broadcast state and tunes to the channel being viewed when the user last powered-off the STB. Basic channel selection to change channels is achieved using the `CH_UP`/`CH_DOWN` keys, by direct channel number entry, or by the `LAST` key (to get to the previously viewed channel).

## Processing Channel Selection

The Navigation Manager API is initialized and either uses a channel ring stored locally in Non-Volatile RAM (NVR), or downloads a default channel ring (DCR). At this point, you can select channels. When you select a channel, the system tunes to the selected channel and displays the selected channel number in the upper-right corner of the safe area for the selection timeout period. If you do not select another channel within this timeout, the last channel becomes the current channel, and the current channel becomes the selected channel. The following table shows the different ways the user can select a channel.

**Table 3-1 Selecting a Channel**

| Selection Method | Description |
|---|---|
| Channel up/down keys | Selects the next higher/lower logical channel number in the default channel ring |
| Last channel (flashback) key | Selects the last channel and swaps the last and current channels |
| Direct numeric entry | Enters only the digits required to select the desired channel. As digits are entered, they display, centered in the safe area. The channel is selected when you press the **Enter** button, or when the selection timeout expires. If you enter all the digits, such as 013 for channel 13, this also effects an immediate channel change. |

# Remote Control Key Usage

The MAUI API allows multiple processes to share access to the remote control. In other words, some applications, such as the player shell, may wish to reserve specific key presses on the remote, hiding them from other applications.

Specifically, there are several keys the player shell always reads:

- Power on/off (standby)
- Interactive Services
- Channel up
- Channel down
- Last Channel Recall

The action that the player shell takes is dependent upon its current operating state. Sometimes the key has no effect. Other times, the effect may vary.

# Sample Player Shell Overview

This section gives an overview of the architecture, responsibilities, and requirements of the sample player shell provided with the DAVID Installation Pak and the DAVID Application Development Pak.

The sample player shell contains a minimal amount of functionality. It is intended as a simple example that can be built upon by the manufacturer.

## Sample Player Shell Responsibilities

The sample player shell responsibilities include:

- Initialize MAUI APIs (the sample player shell assumes that the MAUI input process was launched at system startup)
- Initialization of the following devices:
  - Graphics device
  - Input device
  - MPEG device (both audio and video)
- Reserves the following remote keys:
  - VIP
  - VDT
  - Power
- Download and fork an application via UpLink

## Network-Dependent Responsibilities

The sample player shell establishes a connection with an UpLink server.

# Download and Execute Application Considerations

- Use UpLink to download an application and launch it.

# Porting the Sample Player Shell

The player shell provided with the DAVID Installation Pak and the DAVID Application Development Pak illustrates one possible implementation of a player shell. You will want to port and implement your own version of the player shell for your specific network/STB deployment. This section provides information on how to port the player shell to your environment.

## Customizing the Player Shell

You must customize the player shell source code to conform to your configuration. In particular, modify the code to reflect the following:

- The network configuration. This includes information about the absence or presence of a network provider, and digital and/or analog broadcast reception.
- The names/behaviors of any devices. This includes the specifics of the data [such as T1, E1, or ATM (Asynchronous Transport Mode)] and control channels, infrared/remote control, overlay graphics subsystem, and software/hardware demultiplexing of MPEG data.
- Input application programming interface (API). Update the section of code that maps MAUI input key codes to application logical key codes.
- Navigational Manager API.
- Other code you may wish to alter; for example downloading an application via UpLink. This requires you to write code using UpLink and `systrap` to download modules from the network provider and fork them.

### For More Information

See ***Using UpLink*** for details about UpLink.

# Making the Player Shell Code

The source code and makefile for the player shell is found in the `$MWOS/SRC/DAVID/PSHELLS/SAMPLE` directory.

The resulting executable module is named `pshell`.

## Object Files

You must place the executable in ROM (or FLASH) on the DAVID system.

You can find a compiled version of the player shell executable module in:

`$MWOS/OS9000/PPC/CMDS/DAVID`

# Chapter 4: Using the DAVID Demo Library

This chapter describes the DAVID Demo API. The functions are used in other DAVID demonstration programs. The functions were designed as examples that the developer can either use as they are, or enhance.

The functions are located in the library `david_demo.l`.

The source code for david_demo.l is located in `$MWOS/SRC/DAVID/DEMOS/LIBSRC/DAVID_DEMO`.

# Introduction to david_demo.l

This chapter describes the `david_demo.l` library. The functions can help a developer more rapidly prototype application demos.

The set of functions in the `david_demo.l` library are not a comprehensive suite of demo functions. They are created as examples of how to accomplish things that may not be obvious at first glance.

## Function Reference

This section provides detailed information about the functions in the library.

**Table 4-1 Function Calls in `david_demo.l`**

| Functions | Description |
|---|---|
| `david_demo_display_image()` | Display an image |
| `david_demo_err()` | Prints an error message |
| `david_demo_get_pids()` | Determine audio/video PIDs for an MPEG stream |
| `david_demo_getdata()` | Return the contents of a file from an UpLink Server |
| `david_demo_load_iff()` | Load an IFF image into a MAUI drawmap |
| `david_demo_open_cc()` | Open Control Channel |
| `david_demo_open_dc()` | Open Data Channel |

**Table 4-1 Function Calls in `david_demo.l`**

| Functions | Description |
|---|---|
| `david_demo_uplink_init()` | Initialize UpLink |
| `david_demo_uplink_term()` | Terminate UpLink |

# david_demo_display_image()

Display an image

## Syntax

```
error_code
david_demo_display_image(
     GFX_DEV_ID   gfxdev,
     GFX_DMAP     *src_dmap,
     GFX_DMAP     **dst_dmap,
     GFX_VPORT_ID *vport
     );
```

## Description

This function is used to display an image that is currently in an off-screen drawmap. This is accomplished by creating a matching drawmap (in coding method and resolution) that the graphics device supports using graphics memory. Then, copying the source drawmap content and palette, creating a viewport for the new drawmap and displaying it.

The value of this function is that you cannot, for example, directly load an image into graphics memory if that memory is not directly accessible.

## Parameters

| | |
|---|---|
| `gfxdev` | Open graphics device ID. |
| `src_dmap` | Pointer to the drawmap currently containing the image to display. |
| `dst_dmap` | Pointer to a drawmap structure that will be initialized by the function. |
| `vport` | Pointer to a viewport ID that will be initialized by the function. |

## david_demo_err()

Prints an error message

### Syntax

```
error_code
david_demo_err(
     char       *function_name,
     error_code error
     );
```

### Description

This function is used print application errors to `stderr`.

### Parameters

| | |
|---|---|
| `function_name` | Pointer to the name of the function that returned an error. |
| `error` | Error code returned by the function. |

# david_demo_get_pids()

Determine audio/video PIDs for an MPEG stream

## Syntax

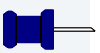
```
error_code
david_demo_get_pids(
     path_id      dp,
     u_int16      *audio_pid,
     u_int16      *video_pid,
     u_int16      *pcr_pid,
     u_int32      timeout
     );
```

## Description

This function returns the audio, video, and PCR PID of the first program found in the MPEG stream currently arriving over the network.

This is accomplished by parsing the next PAT and the next occurrence of the PMT specified as the first program in the PAT.

If a value of zero is returned as the PCR PID, it indicates that the PCR data is included with the video PID stream.



### Note

Not all MPEG-2 transport streams are encoded correctly. For example, the stream davidpr.mp2 has a valid PAT and PMT at the beginning of the stream, but the remaining PMTs are invalid. Since this code is executed in user state mode, it is not fast enough to get the first PAT, parse it and request the PMT PID before the first (valid) PMT has already passed. Fortunately, this problems seems limited to streams that were encoded as MPEG-1 streams and then post-processed to add the MPEG-2 transport layer.

### Parameters

| | |
|---|---|
| `dp` | Open path to the network device. |
| `audio_pid` | Pointer to where the audio PID found will be stored. |
| `video_pid` | Pointer to where the video PID found will be stored. |
| `pcr_pid` | Pointer to where the PCR PID found will be stored. |
| `timeout` | The number of seconds to wait for a PAT or PMT to arrive. |

# david_demo_getdata()

Return the contents of a file from an UpLink Server

## Syntax

```
error_code
david_demo_getdata(
     char          **ret_buff,
     char          *file_name,
     u_int32       *ret_buff_sz,
     int           flag
     );
```

## Description

This function is used to download a file via the UpLink API into a buffer. The function will allocate a buffer large enough to hold the entire contents of the file.

If the function is successful, it is the responsibility of the caller to call either `free()` or `systrap_free()` for the allocated buffer.

## Parameters

| | |
|---|---|
| `ret_buff` | The address of a pointer variable in which a pointer to the buffer allocated for the file will be stored. |
| `file_name` | The name of the file to download from the UpLink server. |
| `ret_buff_sz` | The amount of memory allocated for the file. |
| `flag` | If `flag` is set to `__USER_STATE_MEMORY`, the buffer is allocated using `malloc()`.<br><br>If `flag` is set to `__SYSTEM_STATE_MEMORY`, the buffer is allocated using `systrap_alloc()`. |

## david_demo_load_iff()

Load an IFF image into a MAUI drawmap

### Syntax

```
error_code
david_demo_load_iff(
     GFX_DMAP     **ret_dmap,
     u_int32      dmap_shade,
     u_int32      pixmem_shade,
     char         *buf
     );
```

### Description

This function will parse an IFF image in memory, create an appropriate drawmap for it, and copy the image into the drawmap.

### Parameters

| | |
|---|---|
| `ret_dmap` | Address of a drawmap pointer which will be allocated. |
| `dmap_shade` | MAUI memory shade to use for the drawmap object and palette. |
| `pixmem_shade` | MAUI memory shade to use for pixel memory. |
| `buf` | Pointer to the IFF image in memory. |

## Syntax

```
error_code
david_demo_open_cc(
     path_id      *control_channel
     );
```

## Description

This function is used to open the control channel device specified by the CDB.

## Parameters

| | |
|---|---|
| `control_channel` | Pointer to a path variable to be assigned the control channel path that is opened. |

## david_demo_open_dc()

## Open Data Channel

### Syntax

```
error_code
david_demo_open_dc(
     path_id      *data_channel
     );
```

### Description

This function is used to open the data channel device specified by the CDB.

### Parameters

| | |
|---|---|
| `data_channel` | Pointer to a path variable to be assigned the data channel path that is opened. |

# david_demo_uplink_init()

Initialize UpLink

## Syntax

```
error_code
david_demo_uplink_init(
     path_id      *control_channel,
     path_id      *data_channel
     );
```

## Description

This function is used to open the control and data channels specified by the CDB and initialize the UpLink API, and verify that the connection is valid.

It calls `david_demo_open_cc()`, `david_demo_open_dc()`, `ul_init()`, and `ul_con_sync()`.

## Parameters

| | |
|---|---|
| `control_channel` | Depending upon `flag`, a pointer to a path variable to be assigned a new control channel path, or, a currently open control channel path. |
| `data_channel` | Depending upon `flag`, a pointer to a path variable to be assigned a new data channel path, or, a currently open data channel path. |

## david_demo_uplink_term()

Terminate UpLink

### Syntax

```
error_code
david_demo_uplink_term(
     path_id      *control_channel,
     path_id      *data_channel
     );
```

### Description

This function is used to terminate the UpLink API, and close the control and data channel paths.

### Parameters

| | |
|---|---|
| `control_channel` | Pointer to the open control channel path. |
| `data_channel` | Pointer to the open data channel path. |

# Index

**G**



**I**



**L**



**M**



**N**

## O



## P



## R



## S

**T**



**U**



**V**

# 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_____