Tapwave® Programmer's Reference

Version 1.1a

Copyright

1.Overview

This document describes the APIs developed by Tapwave for Tapwave devices. These APIs support the unique hardware features in Tapwave devices, and assist developers in writing high performance, interactive games.

Developers can use APIs from a variety of sources to create applications, including the Fathammer X-Forge™ 3D Game Engine, Fathammer X-Forge™ Core gaming APIs, standard Palm OS® APIs, as well as the custom Tapwave APIs described in this document. See the appropriate documentation for descriptions of the other APIs.

- **Tapwave Platform APIs:** Tapwave Developer's Overview
- **Fathammer X-Forge 3D Game Engine & Fathammer X-Forge Core APIs:** X-Forge 1.0.2 Guide
- **Palm OS APIs:** Palm OS Programmer's API Reference

2.Specifications

2.1.Description

2.1.1.Large Screen and Landscape Support

Tapwave devices have a large screen, which can be used in both landscape and portrait modes. The default configuration is 480x320 pixels in landscape mode. The current Palm OS version 5.x does not directly support these features, however they are supported with the PINS 1.1 APIs from PalmSource. Palm OS 6.0 will support a large screen, but it does not currently support landscape mode. Therefore Tapwave will continue to support the current PINS 1.1 APIs on future Tapwave devices, regardless of the OS used.

Tapwave is also providing two additional features that are compatible with Palm OS 6.0: a subset of the PINS 1.1 APIs which controls hiding and showing the Pen Input Area (sometimes referred to as the Graffiti® 2 Area), rotating the screen, and showing and hiding the Status Bar.

Applications can use these APIs to change screen bounds and screen mode dynamically, and to create new windows and forms after the desired mode is enabled. Existing Palm OS APIs are compatible with the newly created windows and forms.

2.1.2.Advanced Input Support

Currently Palm OS has a very simple event model for handling user inputs – the OS generates one event for every button press or pen move. This event model cannot

handle advanced user input devices, such as analog joysticks or Tapwave device's navigator. Typical joysticks have several analog axes and many buttons. Games may use complicated button combinations for different actions. Presenting the full state of the navigator, using Palm OS's current event model, requires multiple events. In addition, existing Palm OS events cannot carry analog input values. Since Palm OS uses different events for different input devices, it is difficult to get the entire input state in a single event. Therefore, Tapwave is introducing a new event model to solve this problem.

The solution is to use a new kind of event queue that provides the complete input state on the device. Instead of mandating a fixed event format, the new event queue uses an application-specified event format and other attributes, which the operating system generates accordingly. For edge-driven inputs, such as button-down or button-up, the operating system generates one event for every transition. For state-driven inputs, such as analog navigator positions or digitizer input, the operating system generates events at an application-specified polling period. The event is formatted according to the application-specified format.

The new event format is very simple. Each event is an array of 32-bit values. Each value in an event represents a specific input requested by the application. For buttons, a zero value means the button is up, and a non-zero value means the button is down. For navigator positions, a value from 32767 to –32767 is returned. For digitizer input, the pen's x and y positions are returned. Applications need to check the pen's z value (pressure) to know whether the pen is pressed. A non-zero z value means the pen is pressed. There are also input features that do not directly match hardware features. For example, a 4-way navigator and an 8-way navigator may be simulated using navigator positions—with a zero value meaning the navigator is in the center position, and a value from 1 to 8 indicating the direction of the navigator: 1 is up, 2 is up-right, and so on, proceeding clockwise. More sophisticated features may be introduced in the future.

Note: Even though they all map to the same hardware, it is possible to retrieve the both the 4-way navigator and 8-way navigator states from one event queue.

2.1.3.2D Graphics Support

For 2D graphics, you can use either a high-level API provided by the X-Forge Core library or a low level API provided by the Tapwave TwGfx library. You can also use the standard Palm OS Window Manager, but it does not take advantage of many of the hardware acceleration features on Tapwave devices.

For an overview and references to additional information, see "2D Graphics" in the *Tapwave Developer's Overview*.

2.1.3.1X-Forge Core 2D

Two-dimensional graphics under X-Forge Core is based mainly on *graphics surfaces*. A graphics surface is a wrapper for the actual pixel data, with additional functionality. More specifically, it is a memory region in the graphic accelerator's memory space

where graphics operations are performed, similar to a device context in Windows® or a Graphics object in Java®. When working with surfaces you can do any of the following:

- Blit surfaces onto surfaces with optional scaling, blending and color keying
- Clip a blitted surface to the target surface automatically
- Fill rectangles and draw lines
- Lock the surface and accessed data directly

For more information, see "Chapter 8, 2D Graphics" in the *X-Forge 1.0.2 Guide.*

2.1.3.2Tapwave TwGfx Graphics API

The Tapwave TwGfx Graphics API allows for low-level access to the graphic accelerator chip functionality, but it still provides an abstraction layer above the chip register set.

Note: Because of limited portability, and possible future incompatibility, Tapwave recommends against using this API. Instead, try to use the X-Forge Core API whenever possible.

The Direct Graphics API provides the following abstractions (*feature groups*):

- **Device -** enumerates the available graphics devices (only one in this case), and initializes the AHI graphics library
- **Surface** - creates, deletes, and manages surfaces
- **Display** - controls features of the display such as: overlays, cursors, bit depth, etc.
- **Draw** - performs rendering operations on surfaces, such as: clipping, stretching, rotation, shading, alpha blending, brush blending, line and glyph drawing, etc.
- **Power** - provides an interface to power the device up and down, and detects the power state

For more information, see the Tapwave TwGfx Graphics API Reference which will be available at a future date.

2.1.4.3D Graphics Support

3D graphics capabilities are provided by the X-Forge Core graphics library. The X-Forge 3D API is very similar to SGI's OpenGL and Microsoft's Direct3D. It is a low-level graphics library, which uses primitives such as triangles, triangle strips, and fans to render 3D objects and scenes. As in OpenGL and Direct3D, the heart of the API is the graphics device. Different graphics devices can render identical graphics calls differently. For example, one device might offer hardware-assisted, perspective-corrected texturing, whereas another device might offer full-scene anti-aliasing.

For more information, see "Chapter 9, 3D Graphics" in the *X-Forge 1.0.2 Guide*.

2.1.5.Bluetooth Collaborative Networking Support

The Tapwave device supports three levels of networking APIs: the standard Palm OS APIs, the X-Forge Core Network API, and the X-Forge Gaming Engine Network API.

For more information, see "Networking" in the *Tapwave Developer's Overview*.

2.1.5.1Palm OS Bluetooth API

Tapwave uses a stack that supports the Bluetooth APIs referenced in the appendix section of the Palm OS 5.1 Reference Manual. This stack is compatible with those APIs.

For comprehensive information on Bluetooth, see "Chapter 6, Bluetooth" in the *Palm OS Programmer's Companion, Volume II, Communications.*

2.1.5.2X-Forge Core Network API

The X-Forge libraries contain an abstraction layer that provides a higher-level model for interactive network gaming. This model runs over either a Bluetooth or a WAN interface, and is accessible via a sockets API. You can choose from four different types of packets: *guaranteed*, *non-guaranteed*, *quick-guaranteed* and *recent-state*. Additionally, a simplified, higher-level networking API is available, which eases development of networked games.

For more information, see "Chapter 5, Overview - Network" in the *X-Forge 1.0.2 Guide*.

2.1.5.3X-Forge Game Engine Network API

The networking API in the X-Forge Game Engine allows packets to be sent from one *client* device to a *receiver* on another device. Receivers are like mail boxes, with an ID number as the address. Receivers allow packets to be sent directly to a specific game object on another device. Each game object typically has its own receiver.

Multiplayer games are implemented by synchronizing game object states over the network. Packets are sent to other devices participating in the same game, and can be sent with various levels of priority.

For more information, see "Chapter 20, Multiplayer Games" in the *X-Forge 1.0.2 Guide*.

2.1.6.Advanced Sound Support

Sound is an important part of the user experience on Tapwave devices, and so it is given more attention in the user interface and API than in generic Palm OS handhelds.

Tapwave has provided some simple wrappers for getting and setting the application volume on the device. All applications which play *primary*sounds (games, music, video) should use these APIs and respect the settings. The user can easily change this volume by pressing and holding the power key. Additional sound controls are provided in the Sound preferences panel.

Palm OS currently provides support for playing PCM/ADPCM sampled sound and simple MIDI tones. Tapwave devices contain advanced hardware that supports playing 128 GM-synthesized instrument/special-effect sounds and 47 drum sounds. Tapwave plans to provide a Direct Sound API to access and play these GM-synthesized sounds. This API will be based on the Yamaha PA-2 Sound Library API. It will provide a MIDI-style interface. *Support for this feature may not appear in the initial Tapwave release.*

The Tapwave volume APIs do set the preferences in Palm OS, so compatible applications can query the game sound level by accessing the `prefSysSoundVolume` System Preference. For more information, see "Chapter 43, Preferences" in the *Palm OS Programmer's API Reference*.

In addition, support for turning the audio amplifier on/off, muting the speakers and headphones, and controlling Bass Boost are available thru the Amplifier Virtual Device API.

2.1.7.Gaming Support

Tapwave provides a gaming support API for recording and reporting High Scores. It should be used to help provide a more consistent and unified gaming experience for the end user. It also enables you to take advantage of some special built-in support, as described below.

2.1.7.1High Score

The Tapwave platform enables you to manage and share high scores for games, stroking the egos of the game players by letting them tell the world of their achievement and compare themselves with other players. There are several pieces needed for this component: APIs and software on the device for the game developers to use, conduits for synchronization to transport the data between the device and Tapwave's website, and finally a web front end to sort, filter, and display the high score data. Two major features are possible: pushing new high scores to the website and pulling others' high scores from the website for local comparison.

The high score manager can be used to store all the high scores for all games on a given device. The application can control how many local scores are saved. Only the highest local score will be sent to the server. The application can request that a certain number of highest scores can be retrieved from the server, but whether this happens or not depends on other variables like whether the user is registered with the website and how often they sync.

There are many different ways of measuring a score and presenting that score to the user. The high score manager, however, understands only one type of score – a 32 bit

unsigned integer. Larger numbers are better scores. This 'normalized' score is never displayed for the user via the high score manager or the website. Instead, a formatted string accompanies each score and this is displayed for the user. A given game can register to track as many different score types (points, time, goals) as it likes, each one is treated as an independent score category. This should give apps the flexibility to report scores however they wish: as times, points, kills, goals, or whatever.

It should not be necessary for the application to include user name or date information in the formatted string, as this data is stored elsewhere with the user account. There is one date included for each high score record, and the user name should come from the device registry not from the game itself.

2.1.7.2Other Useful Functions

Tapwave includes a few other functions which may be of use for Tapwave-specific applications:

- **CtlSetFrameStyle.**This API allows applications to change the frame style of UI-layer controls. It is most useful for turning off the frame on graphic controls. When you use this API in combination with transparency in the graphics, it effectively allows you to create non-rectangular controls.
- **WinGetBitmapDimensions.**This API allows applications to get the effective dimensions of any bitmap (single or double density) in the current coordinate system. It is most useful for centering or right-justifying an image.
- **TwGetSlotNumberForVolume**.This API returns the slot number where the card that contains a VFS volume is inserted. (For VFS volumes which do not belong to any card it returns 0.)
- **TwCreateDatabaseFromImage.**, This API is similar to DmCreateDatabaseFromImage which installs a Palm OS database from a .prc or .pdb format byte stream, but it also accepts byte streams that are compressed with gzip, e.g. a .prc.gz or .pdb.gz file.
- **TwGetGraphicForButton.**, This API returns a small image of a gaming button on the screen. It is useful for creating on-screen documentation or preference controls that enable the user to remap a button.

2.1.8. Vibration Support

Tapwave devices have a built-in *rumbler*. An API gives applications the ability to activate the rumbler device and shake things up a bit. This API provides different types of feedback to the user by specifying the duration and period of the rumble. It takes a stream of bytes, which specifies how long the rumbler is activated, and how long it is quiet. You can define any number of canned effects by varying these intervals, and by repeating the stream of bytes continuously for a certain period. Examples of effects include an engine idling, a shock from an explosion, or driving a vehicle over different road surfaces. The application can then choose from these effects during game play, providing a more realistic, fun experience. The rumbler can also be used by applications to get the user's attention, through the Palm OS Attention Manager. Tapwave-specific constants are defined to address the rumbler.

2.1.9. Digital Rights Management Support

See the *Digital Rights Management* document for details.

3. API Specifications

3.1. Tapwave Features

Palm OS applications that use Tapwave-specific APIs should check the Tapwave API version before they actually call them. The version information is published through the Palm OS feature manager. Applications can use the following sample code to do the check.

```
Boolean TwCheckAPIVersion(UInt32 minVersion) {
    UInt32 version;
    FtrGet(twFtrCreator, twFtrAPIVersion, &version);
    return version >= minVersion;
}
```

At compile time, the Tapwave API version is specified by the constant TAPWAVE_API_VERSION.

```
/* Tapwave API creator */
#define twFtrCreator           'Tpwv'



/* Tapwave API feature number */
#define twFtrAPIVersion        0x0000



/* Current Tapwave API version, which is incremented upon every
update. */
#define TAPWAVE_API_VERSION    ????
```

3.2. Tapwave Virtual Characters

```
/*
 * This vchr is posted when a Tapwave input queue enqueues an
event.
 * In order to do so, applications need to set input queue
capacity to more than
 * zero and activate the queue. Otherwise, an event will not be
enqueued.
 * If an application needs to block for an event, it should use
standard
 * Palm OS EvtGetEvent(), and look for the vchrTwInput. Tapwave
input queues
 * are non-blocking. If an application is not interested in
queued events,
 * it can use TwInputPoll() to find current the input state
without using events.
 */
#define vchrTwInput             (vchrTapwaveMin + 0)



/* Virtual characters for special buttons. */
#define vchrBluetooth           (vchrTapwaveMin + 2)
/* Virtual characters for gaming buttons */
#define vchrFunction            (vchrTapwaveMin + 3)
/* Virtual characters for left and right triggers. */
#define vchrTriggerLeft         (vchrTapwaveMin + 4)
#define vchrTriggerRight        (vchrTapwaveMin + 5)
```

```
/* Virtual characters for action buttons (A top, rest clockwise).
*/
#define vchrActionA             (vchrTapwaveMin + 6)
#define vchrActionB             (vchrTapwaveMin + 7)
#define vchrActionC             (vchrTapwaveMin + 8)
#define vchrActionD             (vchrTapwaveMin + 9)
/*
 * Palm OS 6.0 defines the following constants for the 5-way
navigator.
 * We use the same values for future compatibility. --hz
 */
#if !defined(vchrRockerUp)
#define vchrRockerUp            0x0132      // 5-way rocker up
#define vchrRockerDown          0x0133      // 5-way rocker down
#define vchrRockerLeft          0x0134      // 5-way rocker left
#define vchrRockerRight         0x0135      // 5-way rocker right
#define vchrRockerCenter        0x0136      // 5-way rocker
center/press
#endif



/* Virtual characters for the 5-way navigator. */
#define vchrNavUp               vchrRockerUp
#define vchrNavDown             vchrRockerDown
#define vchrNavLeft             vchrRockerLeft
#define vchrNavRight            vchrRockerRight
#define vchrNavSelect           vchrRockerCenter



/* Virtual characters for 9-way navigator corners.
 * NOTE: These are never returned by EvtGetEvent().
 * Only the 5-way edges are returned by EvtGetEvent().
 * This results in better accuracy with old
 * legacy apps that only understand the 5-way model.
 * 9-way apps, which want the corners, must check
 * the corners after receiving a 5-way keyDownEvent
 * using KeyCurrentState() as follows:
 *
 * Int32 keyState = KeyCurrentState();
 * if ((keyState & keyBitsNavUpLeft) == keyBitsNavUpLeft)
 *    keyCode = vchrNavUpLeft;
```

```
 *
 * The codes below are suggested as "safe" key codes.
 */
#define vchrNavUpLeft           (vchrTapwaveMin + 11)
#define vchrNavUpRight          (vchrTapwaveMin + 12)
#define vchrNavDownLeft         (vchrTapwaveMin + 13)
#define vchrNavDownRight        (vchrTapwaveMin + 14)
```

3.3. Tapwave Error Codes

Tapwave APIs do not define their own error codes. Instead, standard POSIX error codes are used. These widely used error codes are documented in <sys_errno.h>.

3.4. Tapwave Screen APIs

There are three screen APIs, which control:

- **Display orientation:** landscape (wide) or portrait (tall)
- **Pen Input Area:** open or closed
- **Status Bar:** visible or hidden

The Pen Input Area, Status Bar, and Screen Orientation APIs are part of the PINS 1.1 API model, which PalmSource has defined as a standard for Palm OS 5.2 licensees. Licensees may implement this model, which will be available and supported by PalmSource in Palm OS 6.0.

Applications can use any part of the display that is not covered by the Status Bar or the Pen Input Area. Applications must be careful to define windows and forms that fit the currently available screen dimension (including orientation). The screen bounds can easily be determined by calling `WinGetBounds(WinGetDisplayWindow(), …)`. When the available window size changes, the `SysNotifyDisplayResizedEvent` notification and `winDisplayChangedEvent` events are generated. Applications should adjust their form sizes accordingly in their event handlers.

Note: Applications that need to be compatible with PINS 1.0 should register to receive the `sysNotifyDisplayResizedEvent` notification and use `EvtAddUniqueEventToQueue` to add their own `winDisplayChangedEvent` to the event queue. PINS 1.0 did not include this event. By adding it uniquely, you guarantee that only one event will be seen even on devices that support the event directly. Applications should do all resizing and layout in response to `winDisplayChangedEvent`.

The Tapwave device implements a simple Status Bar. It is located on the edge of the display, either to the far right or far left when in landscape mode, or at the bottom when in portrait mode. The *handedness* control in the General preference panel allows the user to specify on which side the Status Bar and Pen Input Area appear when visible. The Status Bar contains a button/icon, which when tapped, causes the screen

orientation to toggle between portrait and landscape modes. The Status Bar also contains a button/icon, which when tapped, causes the Pen Input Area to be hidden or visible. The application can control whether the Status Bar appears. If the application hides the Pen Input Area, and hides the Status Bar, as described above, then it can use the entire screen. Note that if the Pen Input Area is visible, the Status Bar is visible – it is not possible to hide the Status Bar and display the Pen Input area simultaneously.

The ability for the user to hide the Pen Input Area could cause problems for existing Palm™ applications that assume that a Pen Input method is always available. In particular, an application could bring up a form that requires user input, even if the user had chosen to hide the Pen Input Area. To solve this problem, the display manager automatically shows the Pen Input Area whenever any old window or dialog is opened. Old windows or dialogs are identified as those which do not call `FrmSetDIAPolicyAttr(formP, frmDIAPolicyCustom)`.

Another way for an application to ensure that the Pen Input Area is always available is to first show it, and then disable the Status Bar's Pen Input Trigger button. An application can also disable the Status Bar button that switches screen orientation. This is done through the `TwOrientationTriggerSetState` and `PINSetInputTriggerState` APIs, as described below.

To make a full-screen application with the PINS 1.1 APIs, an application needs to do the following:

Upon launching an application, the Pen Input Area and Status Bar are visible, and the display bounds are 160x160. The screen orientation will most likely be landscape, because the Tapwave launcher displays in landscape mode. However, the system does not guarantee this and, in fact, the screen orientation reflects its previous state.

During `frmLoadEvent` or `frmOpenEvent` (before the first call to `FrmDrawForm`), an application should:

Call `FrmSetDIAPolicyAttr(formP, frmDIAPolicyCustom)` to disable automatic handling of the Pen Input Area state

Call `SysSetOrientation`, `PINSetInputAreaState`, `StatShow/StatHide`, `SysSetOrientationTriggerState`, and `PINSetInputTriggerState` in whatever combination desired to get the screen into the desired configuration. Permissible values for `PINSetInputArea` include open, closed, and user. The user state is defined as the last state the user chose by tapping the Status Bar button. Permissible values for `SysSetOrientation` on Tapwave devices include portrait, landscape, and user. The user state is defined as the last state the user chose for this application by tapping the Status Bar button.

After configuring the screen, the application can call `WinGetBounds(WinGetDisplayWindow(), …)` or `WinGetDisplayExtent(…)` to determine the size of the application area.

Resize the form with `WinSetWindowBounds(FrmGetWindowHandle(formP), …)`

Move and resize form objects as necessary with `FrmGetObjectBounds` and `FrmSetObjectBounds` or `FrmGetObjectPosition` and `FrmSetObjectPosition`.

Finally, the application should display the form with `FrmDrawForm`. Displaying the form also updates the Pen Input Area and Status Bar display.

During `winDisplayChangedEvent` handling, an application should:

Validate that its form is the active form with `FrmGetActiveForm`. (Defensive coding.)

Call `WinGetBounds(WinGetDisplayWindow(), …)` or `WinGetDisplayExtent(…)`to determine the size of the application area.

Resize the form with `WinSetWindowBounds(FrmGetWindowHandle(formP), …)`

Move and resize form objects as necessary with `FrmGetObjectBounds` and `FrmSetObjectBounds` or `FrmGetObjectPosition` and `FrmSetObjectPosition`.

If the form changed size or objects changed location, then redraw the form by calling `FrmDrawForm`. (Try to avoid redrawing if nothing changed.)

Return handled = true from the event handler.

During `winEnterEvent`, the application regains control after some dialog box, other OS, or 3rd party software has temporarily taken control. The OS attempts to restore the Pen Input Area, Status Bar, orientation, and trigger states for the application. However, because some software is not smart about the Pen Input Area, Status Bar, and so on, the screen may be left in an unexpected state. To correct this, the smart application can optionally:

Validate that the `winEnterEvent` corresponds to the correct application form, by calling either `FrmGetActiveForm` or `FrmGetActiveFormID`, as appropriate. (Defensive coding.)

Call `SysSetOrientation`, `PINSetInputAreaState`, `StatShow/StatHide`, `SysSetOrientationTriggerState`, and `PINSetInputTriggerState` in whatever combination desired to get the screen into the desired configuration. Note that if the user state is selected, the actual user choice may be different from the previous one due to the user changing the state.

Note that if any state change was made, a new `winDisplayChangedEvent` is sent, and the application can do all the form sizing and object moving in response to that event. Applications do not need to resize in response to `winEnterEvent`.

During application exit:

It is **not** required or recommended to restore any of the Status Bar, Pen Input Area, orientation, or trigger states. The Tapwave implementation of the PINS specification restores all of these states (except orientation) to their default before launching the next application. Just quit.

Applications that wish to work with other devices that implement the 1.0 version of the PINS specification need to be aware of limitations in that model, and how to work around them, as follows:

The `winDisplayChangedEvent` is not sent by PINS 1.0 implementations. Applications can register to receive the `sysNotifyDisplayResizedEvent` notification, and can respond by adding the `winDisplayChangedEvent` using the 'unique' function to avoid duplicate events:

```
EventType event;
MemSet(&event, sizeof(EventType), 0);
event.eType = winDisplayChangedEvent;
EvtAddUniqueEventToQueue(&event, 0, true);
```

Be aware that the PINS 1.0 implementation may require you to call `WinSetConstraintsSize` before making any other PINS function call, and it may require you to make these calls during `frmLoadEvent` handling. The PINS 1.1 specification does not require the use of `WinSetConstraintsSize`.

Be aware that the PINS 1.0 specification does not allow the `pinInputAreaUser` mode for `PINSetInputAreaState`. Instead, the initial Pen Input Area state may be left in the last user state. Tapwave attempts to support this by watching for applications that call `FrmSetDIAPolicy` but do not explicitly call `PINSetInputAreaState`. In these cases, the system sets the Pen Input Area to the last user state. Unfortunately, this happens during the first call to `FrmDrawForm`. So on Tapwave devices, the applications that do this initially open in a small state and then quickly resize if necessary.

3.4.1.Getting Hardware Characteristics

There are some features in the PINS 1.1 specification which you can use to determine the orientation, Pen Input Area, and Status Bar capabilities of a device.

3.4.1.1PINS version

The version of the PINS implementation is available in the feature specified by `pinCreator` and `pinFtrAPIVersion`. Example:

```
UInt32 pinVersion;
FtrGet (pinCreator, pinFtrAPIVersion, &pinVersion);
if (pinVersion >= pinAPIVersion1_1)
     // PINS 1.1 APIs are available
```

3.4.1.2Pen Input Area Capabilities

The capabilities (if any) of the Pen Input Area on a device that supports the PINS 1.1 specification can be read from the feature specified by `sysFtrCreator` and `sysFtrNumInputAreaFlags`:

```
UInt32 inputAreaCapabilities;
FtrGet (sysFtrCreator, sysFtrNumInputAreaFlags, &
inputAreaCapabilities);
if (inputAreaCapabilities & grfFtrInputAreaFlagDynamic)
      // has dynamic input area
if (inputAreaCapabilities & grfFtrInputAreaFlagLiveInk)
      // input area shows feedback when writing
if (inputAreaCapabilities & grfFtrInputAreaFlagCollapsible)
      // input area can be closed
if (inputAreaCapabilities & grfFtrInputAreaFlagLandscape)
      // input area supports landscape orientation
if (inputAreaCapabilities & grfFtrInputAreaFlagLefthanded)
      // input area supports lefthanded preference
```

3.4.2.Screen Orientation

SysGetOrientation

Purpose	Get current screen mode (portrait or landscape).
Prototype	`UInt16 SysGetOrientation (void);`
Result	The current screen orientation, either `sysOrientationPortrait` or `sysOrientationLandscape.`
Comments	This function returns the current screen orientation, either landscape or portrait. It does not have an error condition.
Header	`PenInputMgr.h (included by Tapwave.h)`

Constants	`sysOrientationPortrait`
	`    Landscape screen mode`
	
	`sysOrientationLandscape`
	`    Portrait screen mode`

Sample	**See sample for SysSetOrientation** API.

SysSetOrientation

Purpose	Set screen orientation to portrait or landscape.
Prototype	`Err     SysSetOrientation(UInt16 orientation)`

Parameters	`[in] orientation`	New screen mode, one of the following: `sysOrientationPortrait`, `sysOrientationLandscape`, or `sysOrientationUser`.
Result	`errNone` `pinErrInvalidParam`	Succeeded. Invalid orientation mode passed.
Side Effects	Newly created windows and forms use the new screen mode.	
Events	`SysNotifyDisplayResizedEvent` notification and `winResizeEvent` event.	
Comments	This function sets the new screen orientation to either landscape (wide) or portrait (tall). The actual screen dimensions depend upon whether or not the Pen Input Area and Status Bar are visible, and on high- versus low-density pixel mode. Applications should use `PINSetInputAreaState` to show/hide the Pen Input Area, and `StatShow/StatHide` to show/hide the Status Bar. In landscape mode, the application window may appear to the left or right of the display if the Pen Input Area is enabled. The actual position is controlled by user preference. Applications should never assume or change this window position. Applications should use `WinGetBounds(WinGetDisplayWindow(), …)` or `WinGetDisplayExtent(…)` to find the actual screen size. Note that additional orientation flags are defined for the reverse orientations. Tapwave devices do not support these orientations.	
Header	`PenInputMgr.h (included by Tapwave.h)`	

Constants	sysOrientationPortrait
	    Set screen orientation to portrait.
	
	sysOrientationLandscape
	    Set screen orientation to landscape.
	
	sysOrientationUser
	    Set screen orientation to whatever the user chose last time.

Sample	WinHandle window;
	Int32 prevMode;
	Err err;
	RectangleType rect;
	
	// Save previous screen mode
	prevMode = SysGetOrientation();
	// Change to portrait mode
	(void) SysSetOrientation(sysOrientationPortrait);
	// Create a full screen window
	WinGetBounds(WinGetDisplayWindow(), &rect);
	window = WinCreateWindow(&rect, noFrame, true, true,
	     &err);
	// Erase entire window
	WinEraseWindow();
	// Any drawing operation
	...
	// Delete window
	WinDeleteWindow(window, false);
	// Restore previous mode
	(void) SysSetOrientation(prevMode);

SysGetOrientationTriggerState

Purpose	Get current state of the Status Bar button that switches screen orientation mode.
Prototype	UInt16 SysGetOrientationTriggerState(void)
Result	The current state of the screen orientation trigger, either sysOrientationTriggerDisabled or sysOrientationTriggerEnabled.

Side Effects	None
Comments	The Tapwave Status Bar contains a button that toggles the screen mode orientation between landscape and portrait modes. This button can be disabled by applications that want to prevent the user from switching orientation mode while the application is running. This API allows the application to determine what the current state of this button is – enabled and functioning, or disabled and not functioning. Note that in the non-functioning case, the button appears grayed-out as an indication to the user.
Header	`PenInputMgr.h (included by Tapwave.h)`

Constants	`sysOrientationTriggerDisabled`
	`Status bar button that switches orientation is currently disabled.`
	
	`sysOrientationTriggerEnabled`
	`Status bar button that switches orientation is currently enabled.`

Sample	**See sample for SysSetOrientationTriggerState** API.

SysSetOrientationTriggerState

Purpose	Enable or disable the Status Bar button that switches screen orientation mode.	
Prototype	`Err SysSetOrientationTriggerState(UInt16 state)`	
Parameters	`[in] state`	New button state, either `sysOrientationTriggerDisabled` or `sysOrientationTriggerEnabled`

Result	`ErrNone` – Succeeded. `pinErrInvalidParam` – invalid state passed.
Side Effects	Enables or disables the functioning of the button on the Status Bar that allows changing the screen mode orientation. If the button is disabled, it appears grayed-out in the Status Bar as an indicator to the user that it is non-functional.
Events	`SysNotifyDisplayResizedEvent` notification and `winResizeEvent` event
Comments	The Tapwave Status Bar contains a button that toggles the screen mode orientation between landscape and portrait modes. This button can be disabled by applications that want to prevent the user from switching orientation mode while the application is running.
Header	`PenInputMgr.h (included by Tapwave.h)`

Constants	`sysOrientationTriggerEnabled`
	`    Enable the Status Bar button that toggles screen orientation.`
	
	`sysOrientationTriggerDisabled`
	`    Disable the Status Bar button that toggles screen orientation.`

Sample	`UInt16 prevMode;`
	
	`// Save previous trigger state`
	`prevMode = SysGetOrientationTriggerState();`
	`// Disable the trigger`
	`(void) SysSetOrientationTriggerState`
	`     (sysOrientationTriggerDisabled);`
	`...`
	`// Restore previous trigger state`
	`(void) SysSetOrientationTriggerState (prevMode);`

3.4.3.Pen Input Area

These functions comprise part of the *PalmSource "Dynamic Input Area"*(PINS 1.1 APIs.) Tapwave has adapted these functions to improve forward- and backward-compatibility with other platforms that implement PINS.

PINGetInputAreaState

Purpose	Return the current state of the dynamic Pen Input Area.
Prototype	`UInt16 PINGetInputAreaState(void);`
Parameters	`None.`
Result	One of the constants defined below.
Comments	Applications call this function to determine the current state of the Pin Input Area.
Header	`PenInputMgr.h`

Constants	`pinInputAreaOpen`
	`    The dynamic input area is visible.`
	
	`pinInputAreaClosed`
	`    The dynamic input area is hidden.`
	
	`pinInputAreaNone`
	`    The devices does not support a dynamic input area. (Never returned on this Tapwave release.)`

Sample	**See sample for PINSetInputAreaState** API.

PINSetInputAreaState

Purpose	Set the state of the input area.
Prototype	`Err PINSetInputAreaState(UInt16 state);`

Parameters	`[in] state`	The state to which the input area should be set (opened or closed).
Result	`ErrNone` – Succeded. `pinErrNoSoftInputArea` – There is no dynamic input area on this device. `pinErrInvalidParam` – You have entered an invalid state parameter.	
Side Effects	Since the Pen Input Area cannot be displayed without the Status Bar being displayed, `PINSetInputAreaState` automatically causes the Status Bar to display if the Pen Input Area is activated from a full screen configuration (i.e., no Status Bar or Pen Input Area). If the Pen Input Area or Status Bar changes state, the notification `sysNotifyDisplayResizedEvent` is broadcast, and a `winDisplayChangedEvent` is enqueued.	
Comments	Applications call this function to open or close the Pen Input Area. The Silkscreen Manager responds to this function by opening or closing the Pen Input Area as requested. Applications that want to use the entire screen need to use this function to close the Pen Input Area and hide the Status Bar using `StatHide`. Applications can determine the actual size of the available window area by calling `WinGetBounds(WinGetDisplayWindow(), …)`. Applications can not draw outside of this application area.	
Header	`PenInputMgr.h`	

Constants	`pinInputAreaOpen`
	`Display the dynamic input.`
	
	`pinInputAreaClosed`
	`Hide the dynamic input area.`
	
	`pinInputAreaUser`
	`Set the dynamic input area to the last state the user chose by tapping the Status Bar button. May be shown or hidden.`	

Sample	**See sample for StatShow** API.

PINGetInputTriggerState

Purpose	Return the state of the Status Bar input area button.
Prototype	`Int16 PINGetInputTriggerState(void);`
Parameters	None.
Result	The current state of the input area trigger (see constants below).
Side Effects	None
Comments	The Tapwave Status bar contains a button that causes the Pen Input Area to appear or disappear. This button can be disabled by applications that want to prevent the user from showing or hiding the Pen Input Area while the application is running. This API allows the application to determine what the current state of this button is – enabled and functioning, or disabled and non-functioning. Note that in the non-functioning case, the button appears grayed-out as an indication to the user.
Header	PenInputMgr.h
Constants	pinInputTriggerEnabled The Status Bar button is enabled, and the user can open and close the dynamic input area. pinInputTriggerDisabled The Status Bar button is disabled and the user cannot open and close the dynamic input area. pinInputTriggerNone This device does not support a dynamic input area (never returned by this Tapwave release.)

Sample	**See sample for PINSetInputTriggerState** API.

PINSetInputTriggerState

Purpose	Set the state of the input area button in the Status Bar.	
Prototype	`Err PINSetInputTriggerState(Int16 state);`	
Parameters	`[in] state`	The state to which the input trigger should be set. See constants below.
Result	`ErrNone` – Succeeded. `pinErrNoSoftInputArea` – There is no dynamic input area on this device. `pinErrInvalidParam` – You have entered an invalid state parameter.	
Side Effects	Enables or disables the functioning of the button on the Status Bar that allows the user to show or hide the Pen Input Area. If the button is disabled, it appears grayed-out in the Status Bar, as an indicator to the user that the button is non-functional.	
Comments	The Tapwave Status Bar contains a button that causes the Pen Input Area to appear or disappear. This button can be disabled by applications that want to prevent the user from showing or hiding the Pen Input Area while the application is running. This API allows the application to enable or disable this button.	
Header	`PenInputMgr.h`	

Constants	pinInputTriggerEnabled The Status Bar button is enabled, and the user can open and close the dynamic input area. pinInputTriggerDisabled The Status Bar button is disabled and the user cannot open and close the dynamic input area.

Sample	Int16 prevMode;
	
	// Save previous trigger state
	prevMode = PINGetInputTriggerState ();
	// Disable the trigger
	(void) PINSetInputTriggerState
	(pinInputTriggerDisabled);
	...
	// Restore previous trigger state
	(void) PINSetInputTriggerState (prevMode);

3.4.4.Status Bar

These functions comprise part of the PalmSource PINS 1.1 APIs.

StatHide

Purpose	Hide the Status Bar.	
Prototype	Err StatHide(void);	
Parameters	None.	
Result	errNone – Succeeded. statErrNoStatusBar – The device does not support a Status Bar. statErrInputWindowOpen – The Pen Input Area is open.	
Pre-Conditions	**The Pen Input Area must be closed (see INSetInputAreaState**	

	).
Side Effects	If the Status Bar changes state, the notification `sysNotifyDisplayResizedEvent` is broadcast, and a `winDisplayChangedEvent` is enqueued.
Comments	This function can be called by an application that needs to draw on the entire display area. **Note**: Tapwave discourages you from hiding the Status Bar, since this prevents the user from receiving and accessing status information. This function has no effect if the Status Bar is already hidden.
Header	`PenInputMgr.h`
Sample	**See sample for StatShow** API.

StatShow

Purpose	Display the Status Bar.
Prototype	`Err StatShow(void);`
Parameters	`None.`
Result	`ErrNone` – Succeeded. `statErrNoStatusBar` – The device does not support a Status Bar.
Side Effects	If the Status Bar changes state, the notification `sysNotifyDisplayResizedEvent` is broadcast, and a `winDisplayChangedEvent` is enqueued.

Comments	This function can be called by an application to display the Status Bar. Use `PINSetInputAreaState` to display the Pen Input Area. This function has no effect if the Status Bar is already hidden.
Header	`PenInputMgr.h`

Sample	`Int16 prevPinMode;`
	
	`// Save current input area state`
	`prevPinMode = PINGetInputAreaState ();`
	`// Take down the PINS area`
	`PINSetInputAreaState(pinInputAreaClosed);`
	
	`// Take down the Status Bar`
	`if(StatHide() != errNone){`
	`  // Handle error`
	`}`
	`// Get size of window area and draw, draw, draw`
	`...`
	`// Bring Status Bar back up`
	`if(StatShow() != errNone){`
	`  // handle error`
	`}`
	`// Restore previous input area state`
	`(void) PINSetInputAreaState (prevMode);`

StatGetAttribute

Purpose	Query the current state of the Status Bar.	
Prototype	`Err StatGetAttribute(UInt16 selector, UInt32* dataP)`	
Parameters	`[in] selector`	Selector for the desired attribute
	`[out] dataP`	Pointer to UInt32, to contain returned the data

Result	`errNone` – Succeeded. `statErrNoStatusBar` - Device does not support a Status Bar. `statErrInvalidName` - Selector is not valid.
Pre-Conditions	The Status Bar must be initialized.
Side Effects	None

Comments	The following selectors are supported:
	
	`#define statAttrBarVisible 0 // Status Bar is visible`
	`#define statAttrDimension 1 // bounds of Status Bar window`
	If `selector` is `statAttrBarVisible`, the value pointed at by `dataP` is set to 0 if the Status Bar is hidden, and set to 1 if the `Status Bar` is showing.
	
	If `selector` is `statAttrDimension`, `dataP` is treated as a pointer to two `UInt16`'s. The first `UInt16` is set to the width of the Status Bar and the second is set to the height of the Status Bar based on the standard Palm OS coordinate system. The most common Status Bar size is 160x14.

Header	`StatusBar.h`
Constants	`#define statAttrBarVisible 0 // Status Bar is visible` `#define statAttrDimension 1 // bounds of Status Bar window`

3.4.4.1Handedness Preference

This new system preference is available to control left- or right-handed preference.

```
UInt32 handedness = PrefGetPreference(prefHandednessChoice);
```

Handedness is now one of `prefRightHanded` or `prefLeftHanded`.

To change the handedness preference, call:

```
UInt32 handedness = prefRightHanded; // or prefLeftHanded
PrefSetPreference(prefHandednessChoice, handedness);
```

Note: In the Tapwave additions to Palm OS 5.2, the handedness preference is NOT part of the system preference structure, and you cannot get or set the value using `PrefSetPreferences` or `PrefGetPreferences`. Use the one-at-a-time singular version of the preferences function instead. For additional information, see "Chapter 43, Preferences" in the *Palm OS Programmer's API Reference .*

3.4.5. Tapwave Input API

Note: All Tapwave Input APIs are *non-blocking*.

TwInputOpen

Purpose	Create an input event queue.	
Prototype	`Err TwInputOpen(TwInputHandle* queue, const Char* name, const Char* mode);`	
Parameters	`[out] queue`	The newly created input event queue.
	`[in] name`	The name of the input queue. Must be `"twinput"`.
	`[in] mode`	The mode of the input queue. Must be "`r`".
Result	`errNone` – Succeeded.	
Post-Conditions	A queue has zero capacity when first created. Applications use the `TwInputSetCapacity` API to specify the number of events it should hold.	

Comments	This function creates an input event queue, which can be used to receive all user input on the device. Unlike most event systems, which use different events for different user input devices, the Tapwave system delivers all user inputs in one notification event. Applications use the `TwInputSetFormat` API to specify the user events they want notification of, and the desired format. The input event queue does not guarantee thread-safety. If an application needs to access an event queue from multiple threads at the same time, it must use proper thread synchronization, otherwise the result is undefined. IMPORTANT NOTES:The Tapwave input interface may be implemented with either the user-level library or kernel-level driver. If the interface is implemented using the library, it may crash upon invalid arguments, such as bad pointers. This behavior is similar to the C runtime library, such as memcpy(). If the interface is implemented using the kernel-level driver, it returns EBADF for invalid handle, EFAULT for invalid memory address, and EINVAL for other invalid arguments. This behavior is similar to UNIX system calls, such as read().
Header	`TwInput.h (included by Tapwave.h)`

Sample	`TwInputHandle queue;`
	`// Create input queue`
	`if (TwInputOpen(&queue, "twinput", "r") != errNone) {`
	`  // error`
	`}`
	`// Close input queue`
	`TwInputClose(queue);`

TwInputClose

Purpose	Close an input event queue.	
Prototype	`Err TwInputClose(TwInputHandle queue);`	
Parameters	`[in] queue`	The event queue to close.
Result	`errNone` – Succeeded.	

Pre-Conditions	It is not necessary to deactivate the input queue before closing it.
Comments	This function closes the event queue. Any pending events are lost. Future user input is sent to the Palm OS event queue, which can be retrieved using the Palm OS EvtGetEvent API. NOTE: Passing an invalid argument to this function may cause the application to crash.
Header	`TwInput.h (included by Tapwave.h)`

Sample	`TwInputHandle queue;`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Close input queue`
	`TwInputClose(queue);`

TwInputActivate

Purpose	Activate an input event queue.	
Prototype	`Err TwInputActivate(TwInputHandle queue);`	
Parameters	`[in] queue`	The event queue to activate.
Result	`errNone` – Succeeded. EBUSY – An input queue is already activated.	

Comments	This function acquires or activates the event queue. All future user input is sent to this event queue instead of the system event queue. There are certain events that are not sent to a Tapwave event queue, such as power switch. Applications should use the Palm OS event queue for these events. At any given time, only one event queue can be active. Trying to activate a second event queue will always fail. Once an event queue is activated, it receives all user input. For each event received by the queue, a key event posts to the Palm OS event queue. This event is a special `keyDownEvent` with following information: `ascii = vchrInput, keyCode = 0, modifiers = commandKeyMask`. IMPORTANT NOTE: An application should only activate the input queue when it owns the focus form, and deactivate the queue immediately when it receives the `winExitEvent`. Otherwise it will block other forms from receiving regular events, which will in fact hang the entire system. **Note:** Passing an invalid argument to this function may cause the application to crash.
Header	`TwInput.h (included by Tapwave.h)`

Sample	`TwInputHandle queue;`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Activate input queue`
	`if (TwInputActivate(queue) != errNone) {`
	`  // error`
	`}`
	`...`
	`// Deactivate input queue`
	`TwInputDeactivate(queue)`
	`// Close input queue`
	`TwInputClose(queue);`

TwInputDeactivate

Purpose	Deactivate an input event queue .
Prototype	`Err TwInputDeactivate(TwInputHandle queue);`

Parameters	[in] queue	The event queue to deactivate.
Result	errNone – Succeeded.	
Comments	This function deactivates the event queue. All future user input is sent to the system event queue. Deactivating an event queue does not remove pending events from the queue. Applications can still use TwInputRead to retrieve pending events, or TwInputPoll to check the current user input state. **IMPORTANT NOTE:** An application should deactivate the queue when it receives winExitEvent. **Note:** Passing an invalid argument to this function may cause the application to crash.	
Header	TwInput.h (included by Tapwave.h)	

Sample	TwInputHandle queue;
	// Create input queue
	TwInputOpen(&queue, "twinput", "r");
	// Activate input queue
	TwInputActivate(queue);
	...
	// Deactivate input queue
	TwInputDeactivate(queue)
	// Close input queue
	TwInputClose(queue);	

TwInputGetPeriod

Purpose	Get the polling period of an input event queue.	
Prototype	Err TwInputGetPeriod(TwInputHandle queue, Int32* milliSeconds);	
Parameters	[in] queue	The event queue to query.

	[out] milliSeconds	The polling period in milliseconds.
Result	errNone – Succeeded.	
Comments	The polling period is only significant to analog input devices, such as a navigator. The buttons and pen typically generate events during state transitions, regardless of the polling period. Applications can choose to use TwInputPoll to poll the current navigator status instead of using automatic polling. **Note:** Passing an invalid argument to this function may cause the application to crash.	
Header	TwInput.h (included by Tapwave.h)	

Sample	`TwInputHandle queue;`
	`Int32 millis;`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Query the polling period`
	`TwInputGetPeriod(queue, &millis);`
	`// Close input queue`
	`TwInputClose(queue);`	

TwInputSetPeriod

Purpose	Set the polling period of an input event queue.	
Prototype	`Err TwInputSetPeriod(TwInputHandle queue, Int32 milliSeconds);`	
Parameters	[in] queue	The event queue to query.
	[in] milliSeconds	The polling period in milliseconds.

Result	`errNone` – Succeeded.
Comments	Set the polling period for analog input devices, such as a navigator. The system polls the analog device at the specified polling period, and generates an event if it is engaged. For most analog devices, the system uses a minimal threshold to avoid spurious events: If the user touches the navigator only slightly, the system does not generate an event. **Note:** If the `milliSeconds` is <= 0, it disables automatic input polling. **Note:** Passing an invalid argument to this function may cause the application to crash.
Header	`TwInput.h (included by Tapwave.h)`

Sample	`TwInputHandle queue;`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Set the polling period to 200 ms`
	`if (TwInputSetPeriod(queue, 200) != errNone) {`
	`  // error`
	`}// Close input queue`
	`TwInputClose(queue);`

TwInputGetCapacity

Purpose	Get the capacity of an input event queue.	
Prototype	`Err TwInputGetCapacity(TwInputHandle queue, Int32* capacity);`	
Parameters	`[in] queue`	The event queue to query.
	`[out] capacity`	The capacity of input event queue.
Result	`errNone` – Succeeded.	

Comments	The default capacity is zero. Applications should use explicit polling for zero-capacity queues. The capacity is measured in number of events, not the actual internal buffer size. **Note:** Passing an invalid argument to this function may cause the application to crash.
Header	`TwInput.h (included by Tapwave.h)`

Sample	`TwInputHandle queue;`
	`Int32 capacity;`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Get the queue's input capacity`
	`if (TwInputGetCapacity(queue, &capacity) != errNone) {`
	`  // error`
	`}`
	`…`
	`// Close input queue`
	`TwInputClose(queue);`

TwInputSetCapacity

Purpose	Set the capacity of an input event queue.	
Prototype	`Err TwInputGetCapacity(TwInputHandle queue, Int32 capacity);`	
Parameters	`[in] queue`	The event queue to set capacity for.
	`[in] capcity`	The new capacity for the input event queue (in number of events).
Result	`errNone` – Succeeded.	

Comments	The default queue capacity is zero. Capacity must be between 0 and 16, inclusive. Creating a deep event queue is usually not necessary, as event queues do not usually fill. Note that if an application cannot handle events promptly, a deep event queue will not help. Instead, emphasis should be placed on improving overall application performance. If `capacity` <= 0, it disables event queuing, but the application can still use `TwInputPoll()` to query current device input state. The application cannot change capacity while an input queue is active. **Note:** Passing an invalid argument to this function may cause the application to crash.
Header	`TwInput.h (included by Tapwave.h)`

Sample	`TwInputHandle queue;`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Set queue capacity to 10 events.`
	`if (TwInputSetCapacity(queue, 10) != errNone) {`
	`  // error`
	`}`
	`// Close input queue`
	`TwInputClose(queue);`

TwInputGetFormat

Purpose	Get the format of an input event queue.	
Prototype	`Err TwInputGetFormat(TwInputHandle queue, Int32* formats, Int32* sizeInBytes);`	
Parameters	`[in] queue`	The event queue to query.

	`[out] formats`	The format buffer of the input event queue.
	`[in/out] sizeInBytes`	The size of the formats argument in bytes. On return, it contains the actual format size. It is possible that the actual format size exceeds the provided buffer's size. In this case, only a partial format is copied into the provided buffer.
Result	`errNone` – Succeeded.	
Comments	The size of the data to be returned in the formats buffer cannot be known ahead of time. Therefore, the you should make a best guess for `sizeInBytes`. If `sizeInBytes` is not big enough, the function will return a value in `sizeInBytes` that is larger than the size of the buffer passed in. In this event, you should reissue the call with a larger buffer. **See the TwInputSetFormat** comments for additional information. **Note:** Passing an invalid argument to this function may cause the application to crash.	
Header	`TwInput.h (included by Tapwave.h)`	
Constants	See Inp later in this guide.	

Sample	`#define NUMFORMATS 32`
	`TwInputHandle queue;`
	`Int32 actualSize;`
	`Int32 size = NUMFORMATS;`
	`Int32 *formats = malloc(size * sizeof(Int32));`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Query the format`
	`TwInputGetCapacity(queue, formats, size, &actualSize);`
	`If (actualSize > size) {`
	`  // formats buffer not big enough, try again`
	`  size = actualSize;`
	`  free (formats);`
	`  formats = malloc(size * sizeof(Int32));`
	`  TwInputGetCapacity(queue, formats, size, &actualSize);`
	`}`
	`// Close input queue`
	`TwInputClose(queue);`

TwInputSetFormat

Purpose	Set the format of an input event queue.	
Prototype	`Err TwInputSetFormat(TwInputHandle queue, Int32* formats, Int32 sizeInBytes);`	
Parameters	`[in] queue`	The event queue to query.
	`[in] formats`	The format buffer of input event queue.
	`[in] sizeInBytes`	The size of format buffer in bytes.
Result	`errNone` – Succeeded.	

Comments	Tapwave input events do not have a fixed format. Each application must specify the format to be used. Tapwave events are composed of arrays of Int32 fields. Each field represents an input feature, such as a button press, mouse position, navigator button, navigator position, digitizer output, or a physical volume control setting. In addition, there are special formats that can be requested such as a time stamp (in milliseconds since system boot) and a sequence number (which increments by one for each event on the device). For a list of input feature constants, see Inp later in this guide. Applications must call this function to set the event format before using the input queue. The format of an input queue can be changed more than once. Whenever the format is changed, the input queue is flushed, and all pending events are discarded. Applications can not change the event format while an input queue is active. `TwInputPoll()`increments the event sequence number by one. **Note:** Passing an invalid argument to this function may cause the application to crash.
Header	`TwInput.h (included by Tapwave.h)`
Constants	See Inp later in this guide.

Sample	`struct MyEvent {`
	`  Int32 joyX;`
	`  Int32 joyY;`
	`};`
	`TwInputHandle queue;`
	`Int32 formats[] = { twInputNavX, twInputNavY };`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Set the input queue format`
	`TwInputSetFormat(queue, formats, sizeof(formats));`
	`// Close input queue`
	`TwInputClose(queue);`

TwInputPeek

Purpose	Peek an input event queue.

Prototype	`Err TwInputPeek(TwInputHandle queue, TwEvent* event, Int32 sizeInBytes);`	
Parameters	`[in] queue`	The event queue to query.
	`[out] event`	(optional) The event buffer.
	`[in] sizeInBytes`	The size of event in bytes.
Result	`errNone` – Succeeded. `EAGAIN` – No event is available. `EINVAL` – The event size does not event format.	
Pre-Conditions	Before an application can call `TwInputPeek()`, it must set event format, set capacity > 0, and activate the event queue. Otherwise, the call will fail. For a zero-sized event queue, applications should use `TwInputPoll()`.	
Comments	Peek at next event. The event is not removed from the queue. If no event is available, `EAGAIN` is returned. The event size must match the event format specified by last `TwInputSetFormat` call. **Note:** Passing an invalid argument to this function may cause the application to crash.	
Header	`TwInput.h (included by Tapwave.h)`	
Constants	See `TwInput.h` for more information.	

Sample

```
struct MyEvent {
  Int32 joyX;
  Int32 joyY;
};
TwInputHandle queue;
MyEvent event;
Int32 formats[] = { twInputNavX, twInputNavY };
// Create input queue
TwInputOpen(&queue, “twinput”, “r”);
// Query the format
TwInputSetFormat(queue, formats, sizeof(formats));
// Activate queue
TwInputActivate(queue);
// Peek event
if (TwInputPeek(queue, &event, sizeof(event)) ==
errNone) {
    // There is data in event buffer.
}
// Close input queue
TwInputClose(queue);
```

TwInputRead

Purpose	Read event from input event queue.	
Prototype	`Err TwInputRead(TwInputHandle queue, TwEvent* event, Int32 sizeInBytes);`	
Parameters	`[in] queue`	The event queue to query.
	`[out] event`	(optional) The event buffer.
	`[in] sizeInBytes`	The size of event in bytes.
Result	`errNone` – Succeeded. `EAGAIN` – No event available. `EINVAL` – The event size does not match the event format.	
Pre-Conditions	Before an application can call `TwInputRead()`, it must set event format, set capacity > 0, and activate the event queue. Otherwise, the call will fail. For a zero-sized event queue, applications should use `TwInputPoll()`.	
Comments	Read next event, and remove it from the queue. If an event is not available, error is returned. The event buffer cannot be NULL, and `sizeInBytes` must match the event format specified by the last call to `TwInputSetFormat`. NOTE: Passing an invalid argument to this function may cause the application to crash.	
Header	`TwInput.h (included by Tapwave.h))`	
Constants	See `TwInput.h` for more information.	

Sample	`struct MyEvent {`
	`  Int32 joyX;`
	`  Int32 joyY;`
	`};`
	`TwInputHandle queue;`
	`MyEvent event;`
	`Int32 formats[] = { twInputNavX, twInputNavY };`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Set the input event queue format`
	`TwInputSetFormat(queue, formats, sizeof(formats));`
	`// Activate queue`
	`TwInputAcquire(queue);`
	`// Read event`
	`TwInputRead(queue, &event, sizeof(event));`
	`// Close input queue`
	`TwInputClose(queue);`

TwInputPoll

Purpose	Poll user input through an event queue .	
Prototype	`Err TwInputPoll(TwInputHandle queue, TwEvent* event, Int32 sizeInBytes);`	
Parameters	`[in] queue`	The event queue to query.
	`[out] event`	(optional) The event buffer.
	`[in] sizeInBytes`	The size of event in bytes.
Result	`errNone` – Succeeded. `EINVAL` – The event size does not match the event format.	

Comments	This function polls all input devices and returns the event using the event queue format. It does not change the event queue status in any way. The event buffer cannot be `NULL`, and the event `sizeInBytes` must match the event format specified by the last call to `TwInputSetFormat`. The input queue does not need to be active. `TwInputPoll()`increments the event sequence number by one. **Note:** Passing an invalid argument to this function may cause the application to crash.
Header	`TwInput.h (included by Tapwave.h)`
Constants	See `TwInput.h` for more information.

Sample	`struct MyEvent {`
	`  Int32 joyX;`
	`  Int32 joyY;`
	`};`
	`TwInputHandle queue;`
	`MyEvent event;`
	`Int32 formats[] = { twInputNavX, twInputNavY };`
	`// Create input queue`
	`TwInputOpen(&queue, "twinput", "r");`
	`// Set the input event queue format`
	`TwInputSetFormat(queue, formats, sizeof(formats));`
	`// Poll event queue`
	`if (TwInputPoll(queue, &event, sizeof(event)) != errNone) {`
	`  // error`
	`}// Close input queue`
	`TwInputClose(queue);`

TwInputControl

Purpose	Control the input devices .	
Prototype	`Err TwInputControl(TwInputHandle queue, Int32 command, void* data, Int32 sizeInBytes);`	
Parameter	`[in] queue`	The event queue to query.

s		
	`[in] command`	The control command.
	`[in/out] data`	The control data.
	`[in] sizeInBytes`	The size of control data.
Result	`errNone` – Succeeded. `ENOSYS` – Not Implemented.	
Comments	This API is reserved for future use.	
Header	`TwInput.h (included by Tapwave.h)`	
Constants	See `TwInput.h` for more information.	

3.4.6. Input Event Constants

Tapwave-platform applications specify the events for which they want notification. The table below is a list of the input feature events applications can request. Call the function `TwInputSetFormat` to request event notifications, and use the constants below in the `formats` argument.

Input Selector	Data Type	Comment
`twInputSequence`	`Int32`	Sequence number of the event. Starts at 0 when the input queue is activated, may roll over and become negative over time.
`twInputTimeStamp`	`Int32`	The system tick counter when the event occurred. May roll over and become negative over time.

twInputPenX twInputPenY	Int32	Pen location in standard coordinate system, limited by screen size and orientation.
twInputPenZ	Int32	Pen pressure, which ranges from 0 (pen fully up) to 0x7FFF (pen fully down). On most systems, only binary pen down information is available, in which case treat 0 as pen up and non-zero as pen down.
twInputNavX twInputNavY	Int32	Joystick positions range from –32767 to +32767. 0 means center position.
twInputNavCircR twInputNavCircTheta twInputNavCircX twInputNavCircY	Int32	Joystick positions in polar coordinates. R is the radius, or offset from center, in the range 0 to +32767. Theta is the angle in degrees from 0 to 359, measured counter clockwise from horizontal to the right. CircX and CircY are the *circularized* X and Y value – that is, precomputed R * cos(Theta) and R * sin(Theta), but done efficiently using only integer math.
twInput4Way	Int32	Joystick position interpreted as 4-way navigator, one of: twNavigatorCenter twNavigatorUp twNavigatorRight twNavigatorDown twNavigatorLeft
twInput8Way	Int32	Joystick position interpreted as 8-way navigator, one of: twNavigatorCenter twNavigatorUp twNavigatorUpRight twNavigatorRight twNavigatorDownRight twNavigatorDown twNavigatorDownLeft twNavigatorLeft twNavigatorUpLeft

twInputActionA twInputActionB twInputActionC twInputActionD	Int32	The state of any of the action buttons. 0 means button is up. Non-zero means button is down. A is the topmost action button, then proceeding clockwise.
twInputTriggerLeft twInputTriggerRight	Int32	The state of the left and right triggers on the device edge. 0 means button is up. Non-zero means button is down.
twInputNavUp twInputNavDown twInputNavLeft twInputNavRight	Int32	The state of the navigator in different directions. 0 means the navigator is in that direction. Non-zero means the navigator is not in that direction.
twInputNavSelect	Int32	Pushing down on the navigator triggers the *select* button. 0 means button is up, non-zero means button is down. (Note that pushing in on the navigator can easily change the state of the twInputNavX and twInputNavY positions.)
twInputFunction	Int32	The function button is near the navigator. 0 means button is up, non-zero means button is down.
twInputPower twInputLaunch twInputBluetooth	Int32	These buttons are not meant for gaming, but reserved for device use. 0 means button is up, non-zero means button is down. For arcade-style games the twInputPower button may be used as a pause button.

3.5.Gaming API

3.5.1.High Score

3.5.1.1Registering a game with the high score server

TwHighScoreRegister

Purpose	Register a new game application with the high score manager.

Prototype	`Err TwHighScoreRegister(`
	`  UInt32 creatorID,`
	`  UInt16 scoreType,`
	`  UInt16 numLocalScoresToKeep,`
	`  UInt16 numServerScoresToKeep,`
	`  Boolean reportScoresToServer,`
	`  Char *gameName)`	

Parameters	`creatorID`	The creator ID for the game application itself, used as the key for tracking scores.
	`scoreType`	The secondary identifier for games that need to keep more than one type of score. Pass 0 if not needed.
	`numLocalScoresToKeep`	How many scores reported on the local device to keep.
	`numServerScoresToKeep`	How many scores from the server to keep.
	`reportScoresToServer`	Upload the high score to the Tapwave score server website during sync, only if the user has registered for that service. Normally true.
	`gameName`	The name of the game as a user-visible formatted string.
Result	`errNone` – Succeeded.	
Pre Conditions	None	
Post Conditions	None	

Side Effects	Registers a game with the high score manager, or updates the settings if the game is already registered. Only one registration exists for a given game. If a game re-registers with fewer scores than the previous registration, any extra scores are deleted.
Comments	A game normally needs to register with the high score manager only once per device, but there is no harm in calling register again with the same settings. Changing the registration settings by calling this function may change the state of other records in the database, e.g. lowering the number of high scores may cause high score records to be deleted.
Header	`TwHighScore.h`

TwHighScoreUnregister

Purpose	Unregister a game application from the high score manager.	
Prototype	`Err TwHighScoreRegister(UInt32 creatorID, UInt16 scoreType);`	
Parameters	`creatorID`	The creator ID for the game application itself, used as the key for tracking scores.
	`scoreType`	The secondary identifier for games that need to keep more than one type of score. Pass 0 if not needed.
Result	`errNone` – Succeeded. `twHighScoreErrNotRegistered` – game and score type not registered	
Pre Conditions	None	
Post Conditions	None	

Side Effects	Removes the game and any high scores from the high score registry. Normally called only when the user is deleting a game forever. Only the scores for the given secondary type are removed, to completely delete all game data this function must be called for every score type.
Comments	This may also be useful when a game needs to reset the state of the high score database, e.g. when installing an update. All existing high score records are removed as a side effect.
Header	`TwHighScore.h`

3.5.1.2Reporting a high score

TwHighScoreType

Purpose	Structure to report high scores to the high score manager.	

Prototype	`typedef struct TwHighScoreTag {`
	`  UInt32 score;`
	`  UInt32 checksum;`
	`  UInt32 dateAndTime;`
	`  UInt16 pad;`
	`  Char *userString;`
	`} TwHighScoreType;`	

Parameters	`score`	A *normalized* field containing the high score, used for sorting.
	`checksum`	A checksum value which can be used to validate the score.
	`dateAndTime`	The time in seconds (as returned from TimGetSeconds) when the score was achieved. In local time zone for the device.
	`pad`	In order for the `userString` pointer to be 4-byte aligned it's necessary to insert 2 bytes of pad after the `DateTimeType`. Write 0's here.

	`userString`	The user visible formatted version of the score. If not needed, it's OK to pass NULL. If you pass a record with an empty string ("") when you register the score, you will get back a NULL `userString` pointer when you read the score.
Comments	The structure is used to report high score data about a particular game using `TwHighScoreReport`. A `UInt32` is provided as a sort key for the high scores, it should be a normalized version of the high score such that larger numbers are better scores.	
Header	`TwHighScore.h`	

TwHighScoreReport

Purpose	Report high score to High Score Manager.	
Prototype	`Err TwHighScoreReport(UInt32 creatorID, UInt16 scoreType TwHighScoreType * highScoreP);`	
Parameters	`creatorID`	Creator ID of the application reporting the high score.
	`scoreType`	The secondary identifier for games that need to keep more than one type of score. Pass 0 if not needed.
	`highScoreP`	Pointer to structure containing the high score data. See **Error! Reference source not found.** above.
Result	`errNone` – Succeeded. `twHighScoreErrNotRegistered` – game and score type not registered	
Pre Conditions	Game must be registered with the high score server.	

Side Effects	Enters a record into the High Score Manager database. May cause existing high scores to be removed if there are more than the max scores to keep already registered. On the next HotSync, the High Score Conduit sends the highest local score and user information to the Tapwave.com server, if the user has registered.
Comments	`highScore` is a pointer to a `TwHighScoreType` structure. In future versions, this data may be signed or encrypted to prevent cheating.
Header	`TwHighScore.h`

3.5.1.3Reading high scores

TwHighScoreSummaryType

Purpose	Structure to get aggregate of high score data in high score manager.	

Prototype	`typedef struct TwHighScoreSummaryTag {`
	`  UInt16 numLocalScoresToKeep;`
	`  UInt16 numServerScoresToKeep;`
	`  Boolean reportScoresToServer;`
	`  UInt8 pad[3];`
	`  UInt32 timeSynced;`
	`  Char *gameName;`
	`  UInt16 numLocalScores;`
	`  UInt16 numServerScores;`
	`  UInt32 highestLocalScore;`
	`  UInt32 lowestLocalScore;`
	`  UInt32 highestServerScore;`
	`  UInt32 lowestServerScore;`
	`} TwHighScoreSummaryType;`	

Parameters	`numLocalScoresToKeep`	The maximum number of local scores to keep, as passed to `TwHighScoreRegister`
	`numServerScoresToKeep`	The maximum number of server scores to keep, as passed to `TwHighScoreRegister`

	reportScoresToServer	Whether or not to report scores to the server, as passed to TwHighScoreRegister
	pad	Padding needed for compatible alignment with ARM
	timeSynced	The time in seconds that the high scores were sent to the desktop via HotSync, for transmission to Tapwave.com. The time is reported in the local time zone for the device. 0 if never synced.
	gameName	A pointer to the name of the game, as passed to TwHighScoreRegister.
	numLocalScores	The number of high scores available in the local pool.
	numServerScores	The number of local high score records that exist from the server.
	highestLocalScore	The highest score in the local pool, as reported with TwHighScoreRegister
	lowestLocalScore	The lowest high score record saved on the device. (Normally there would be no need to register a new high score if it was below this value).
	highestServerScore	The best score reported to the server (as of the last sync date).
	lowestServerScore	The lowest score saved from the server (as of the last sync date).

Comments	The structure is used to get information about the high scores for a particular game (and score type) from the database. It provides a quick summary and is used for successive calls to `TwHighScoreGetDetails`.
Header	`TwHighScore.h`

TwHighScoreGetSummary

Purpose	Get info about the current high scores registered with the high score manager.	
Prototype	`TwHighScoreSummaryType *TwHighScoreGetSummary (Uint32 creatorID, UInt16 scoreType);`	
Parameters	`creatorID`	Creator ID of the application for which we want the last high score.
	`scoreType`	The secondary identifier for games that need to keep more than one type of score. Pass 0 if not needed.
Result	Pointer to a newly allocated chunk containing the high score summary. The memory is owned by the calling application, and must be disposed of with `MemPtrFree` when done. The `gameName` pointer within the returned structure points to memory that is within the chunk returned, so one call to `MemPtrFree` deletes both the structure and the string. If game and score type is not registered, the function returns NULL.	

Comments	Use this function to get overall info about the data contained in the high score manager. Use `TwHighScoreGetDetails` to get details about individual high scores after calling this function. `HighScoreSummaryP->numLocalScores` will never be larger than the `numLocalScoresToKeep` passed in the most recent call to `TwHighScoreRegister`. `HighScoreSummaryP->numServerScores` will never be larger than the `numServerScoresToKeep` passed in the most recent call to `TwHighScoreRegister`. The server high score count, high and low scores, and time synced values will all be 0 if there are no server scores available. This may happen if the user has never successfully synced with the Tapwave server or if the user has not registered with the server.
Header	`TwHighScore.h`

TwHighScoreGetDetails

Purpose	Get a high score record from the database.	
Prototype	`TwHighScoreType *TwHighScoreGetDetails (Uint32 creatorID, UInt16 scoreType, ScorePoolType which, UInt16 index);`	
Parameters	`creatorID`	Creator ID of the application for which we want the high score.
	`scoreType`	The secondary identifier for games that need to keep more than one type of score. Pass 0 if not needed.
	`which`	`twHighScorePoolLocal` or `twHighScorePoolServer`.
	`index`	The index number for which score to get, from 0 to `countServerScores-1` or `countLocalScores-1`. If the index is out of range NULL.

Result	Points to a newly allocated (with `MemPtrNew`) chunk that contains a `TwHighScoreType` structure with the data. The string pointers in the structure will point to memory in the newly allocated chunk. The calling application is responsible for calling `MemPtrFree` to dispose of this memory when it is done. If game and score type is not registered, the function returns NULL.
Side Effects	Allocates memory for the new score record.
Header	`TwHighScore.h`

Sample	`TwHighScoreType *scoreP = NULL;`
	`UInt32 goal;`
	
	`if (scoreP == TwHscGetScore(myCreator, 0,`
	`TwHighScorePoolLocal , 0)) {`
	`goal = scoreP->score; // highest score!`
	`MemPtrFree(scoreP);`
	`}`

3.5.1.4

3.5.1.4Tournament Scores

TwHighScoreGetTournament

Purpose	Structure to get aggregate of high score data in high score manager.	
Prototype	`UInt16` **`TwHighScoreGetTournament`**`(UInt32 creatorID, UInt16 scoreTypeIn, Char *code);`	
Parameters	`creatorID`	The creator ID of the game.

	`scoreTypeIn`	If >1, pre-loads the tournament code with the right value for this score type. If 0, uses the passed code as the initial value and prompt the user. If 1, just decodes the passed code but does not run any UI.
	`code`	An array of 4 Char values which holds the initial code on entry, and which is updated to contain the user code on exit. (Note that this may be changed even if the user does not enter a valid code and cancels.)
Result	The `scoreType` that the user enters or which is decoded based on the code chars and `creatorID`. Returns 0 (invalid tournament code) if the user cancels or the code is invalid.	
Comments	This multi-purpose function is used to run the tournament code UI. Normally it is called with a code array set to all 0's and a `scoreTypeIn` of 0. In this mode, the user is prompted to enter the tournament code, and valid tournament codes are translated to a non-zero score type and returned. If an invalid tournament code is entered, an error is presented to the user and they will correct the code. The user may also cancel the dialog, in which case 0 is returned. This function can also be used to decode tournament codes that are gathered via some game-specific UI. To do this, pass 1 as the `scoreTypeIn`, and load the code array with the alphanumerics gathered elsewhere. Any letter or number is allowed, lowercase letters are treated as uppercase. Letter 'i' is treated as number '1', and letter 'o' is treated as number '0'. Finally, this function can be used to generate the tournament code for a given score type. To do this, pass `scoreTypeIn > 1`. In this mode, the initial code value is ignored, and instead the UI is pre-loaded with the tournament code for the given `scoreType`. The rest of the UI runs normally, and on return (assuming the user hits Done), the result matches `scoreTypeIn` and the code array has been filled with the normalized tournament code. (You may wish to use the high score test application to search for interesting tournament codes for your game.)	
Header	`TwHighScore.h`	

3.5.2.Other Useful Functions

3.5.2.1Creating Attractive Graphic Controls

One limitation of the Palm OS UI layer has been that it is challenging to create really nice-looking graphic controls without implementing them entirely as gadgets. Tapwave has updated the UI layer to make it a little bit easier. You can use the function `CtlSetFrameStyle` to change the style of a control's border. The most useful frame style is `noFrame`. Graphic controls with no frame and that provide both a normal and a selected graphic are further modified to never erase the area under the control's bounding rectangle. The result is that with appropriate transparency in the graphics, you can create buttons that are non-rectangular. (Although the active area for hit testing and pen interaction will always be a rectangle.)

CtlSetFrameStyle

Purpose	Change the frame style of a button control.	
Prototype	`void` **`CtlSetFrameStyle`**`(ControlType *ctlP, ButtonFrameType newStyle)`	
Parameters	`ctlP`	A pointer to the control object to modify.
	`newStyle`	The new style for the frame, mostly likely `noButtonFrame`.
Comments	Change the control frame attributes to match the new value. This does not draw the control or change the visibility or any other control attribute. It is most commonly used during form initialization before the form has drawn. This function can be called from native or 68K applications.	
Header	`TwOSAdditions.h, see also Control.h`	

Sample	`ControlType *ctlP;`
	`ctlP = FrmGetObjectPtr( frm, FrmGetObjectIndex( frm, SampleGraphicPushButton ));`
	`CtlSetFrameStyle(ctlP, noButtonFrame);`	

3.5.2.2

3.5.2.2Getting Correct Bitmap Dimensions

With many different bitmap formats and family types, it can often be challenging to figure out just how big a bitmap really is.

WinGetBitmapDimensions

Purpose	Return the width and height of any bitmap, or any bitmap family, in the current coordinate system.	
Prototype	`void` **`WinGetBitmapDimensions`**`(BitmapType *bmP, Coord *widthP, Coord *heightP)`	
Parameters	`bmP`	A pointer to any bitmap or bitmap family.
	`widthP`	(Output) A pointer to a value filled in with the bitmap's effective width in the current coordinate system..
	`heightP`	(Output) A pointer to a value filled in with the bitmap's effective height in the current coordinate system.
Comments	Returns the width and height of any bitmap (68K or ARM) in the current coordinate system. Can be called from native or 68K code.	
Header	`TwOSAdditions.h`	

3.5.3.Determining the location of VFS Volumes

In order to provide attractive user interfaces, it is often useful to have a physical indication of where a VFS volume resides. For example, a volume on a RAM card in the left slot might have a different icon from one in the right slot, which might be different again from a volume mounted on a network file system or on internal memory.

TwGetSlotNumberForVolume

Purpose	Return the slot number for a given VFS volume.	
Prototype	`Int16` **`TwGetSlotNumberForVolume`**`(UInt16 volRef)`	
Parameters	`volRef`	A volume reference number as returned from `VFSVolumeEnumerate`.
Result	0 – Volume is not stored on a card (network, internal, etc.) 1 – Volume is stored on card 1. 2 – Volume is stored on card 2.	
Comments	Returns the slot number for a given volume. Depends on the VFS implementation properly filling out the `VFSVolumeInfo` structures and on Tapwave's slot drivers. Note that in the future, some cards may contain more than one volume. Therefore, code should not assume that finding one volume on a card slot means there are no others. This function can be called from 68K or native applications.	
Header	`TwOSAdditions.h`	

3.5.3.1Installing Databases from Compressed Images

This simple function combines the gzip library with `DmCreateDatabaseFromImage` to provide an easy way to uncompress and install a Palm OS database into the storage heap.

TwCreateDatabaseFromImage

Purpose	Uncompress a gzip Palm OS database and installs it in the storage heap.	
Prototype	`Err` **`TwCreateDatabaseFromImage`**`(void *imageP)`	
Parameters	`imageP`	A pointer to a chunk of memory containing the (probably compressed) .prc or .pdb byte stream.

Result	`errNone` – Succeeded. Other errors per `DmCreateDatabaseFromImage`
Comments	This function works exactly like `DmCreateDatabaseFromImage`, except that it can also handle binary files compressed with gzip. It expands the binary into the dynamic heap before installing it, so there is a limit on the largest database that can be installed with this method. This function is most useful for reducing the amount of data that is moved to the device when installing a new application or other data file. This function can be called from 68K or native applications.
Header	`TwOSAdditions.h`

3.5.3.2Showing Images for Hardware Buttons

Tapwave has created some standard graphics that can be used in the user interface where images of the gaming buttons are required, e.g. for control customization interfaces.

TwGetGraphicForButton

Purpose	Return an image that can be used to help describe a hardware button.	
Prototype	`const BitmapType *TwGetGraphicForButton(WChar chr, Int32 size)`	
Parameters	`chr`	The character code that the button produces when pressed, see the file `TwChars.h` for a list. Note that not all characters have graphics.
	`size`	The size of the graphic to return. Note that only a very small set of sizes is supported.
Result	A pointer to a read-only bitmap indicating the graphic button.	
Comments	Note that this function returns a bitmap pointer, not a resource number, and so you must draw the image yourself with `WinDrawBitmap`.	

Header	TwOSAdditions.h

3.6.Device APIs

3.6.1.Attention Manager Extensions

The following constants will be added to allow the Extensions Manager to use the Tapwave device rumbler to get the user's attention.

3.6.1.1New AttnFlagsTypes

```
kAttnFlagsVibrateBit        0x0004     Triggers vibration (for
compatiblity)
```

3.6.2.General Virtual Device Interface

In order to provide general support for simple hardware on the Tapwave device, such as the rumbler, Tapwave has defined a virtual device model. This model is controlled through simple open, close, read, write, `getProperty`, and `setProperty` calls, which are defined below. Devices are named by string constants defined in the system header `TwVirtualDevice.h`. For the Zodiac device, only the rumbler is defined.

TwDeviceOpen

Purpose	Open a virtual device.	
Prototype	`Err TwDeviceOpen (TwDeviceHandle* handle, const char* name, const char* mode);`	
Parameters	`[out] handle`	Pointer to `TwDeviceHandle` descriptor used in other calls to identify the device opened.
	`[in] name`	Character string name of the device to open.
	`[in] mode`	Mode to open device: "r" read, "w" write, "rw" read and write.

Result	`errNone` – Succeeded. For return values other than `errNone`, see the system header file `sys_errno.h`.
Comments	Open a virtual device, identified in `name`, using the access specified in `mode`. `handle` is initialized on return, and is used in subsequent calls to identify the device opened.
Header	`TwDevice.h (included by Tapwave.h)`

TwDeviceClose

Purpose	Close a virtual device.	
Prototype	`Err TwDeviceClose (TwDeviceHandle handle);`	
Parameters	`[in] handle`	A `TwDeviceHandle` descriptor identifying the device to close.
Result	`errNone` – Succeeded. For return values other than `errNone`, see the system header file `sys_errno.h`.	
Comments	Close the virtual device associated with the `TwDeviceHandle handle`, which was opened by a prior `TwDeviceOpen` call. Shut down the device and free up any associated resources	
Header	`TwDevice.h (included by Tapwave.h)`	

TwDeviceRead

Purpose	Read data from a virtual device.	
Prototype	`Err TwDeviceRead (TwDeviceHandle handle,` `void* buf, Int32* len);`	
Parameters	`[in] handle`	A `TwDeviceHandle` descriptor to read from, which was opened by a prior `TwDeviceOpen` call.
	`[out] buf`	Pointer to buffer to return the data read from the device.

	[in/out] len	Length of buffer passed in. On return, the number of bytes actually read.
Result	errNone – Succeeded. For return values other than errNone, see the system header file sys_errno.h.	
Comments	Read data from device handle, store in buffer buf of maximum size len, returning the actual number of bytes read, or a negative number if an error occurred. Device must have been opened as "r" or "rw".	
Header	TwDevice.h (included by Tapwave.h)	

TwDeviceWrite

Purpose	Write data to a virtual device.	
Prototype	Err TwDeviceWrite (TwDeviceHandle handle, const void* buf, Int32* len);	
Parameters	[in] handle	A TwDeviceHandle descriptor identifying the device to which it is writing.
	[in] buf	Pointer to data to be written to device.
	[in/out] len	Number of bytes to write to device from buf. On return, number of bytes actually written.
Result	errNone – Succeeded. For return values other than errNone, see the system header file sys_errno.h.	

Comments	Write data to the virtual device. The data format is device-specific. The actual number of bytes written is returned in `len`, which initially contains the size of the `buf`. The device must have been opened as writeable, using modes "w" or "rw".
Header	`TwDevice.h (included by Tapwave.h)`

TwDeviceGetProperty

Purpose	Get the specified property value for the device.	
Prototype	`Err TwDeviceGetProperty (TwDeviceHandle handle, Int32 property, void* buf, Int32* len);`	
Parameters	`[in] handle`	`TwDeviceHandle` descriptor identifying the device from which to get a property.
	`[in] property`	Multi-character integer identifying the property whose value is to be retrieved.
	`[out] buf`	Pointer to buffer in which to store the property's value.
	`[in/out] len`	Size of `buf`, in bytes. On return, it contains the actual size of the specified property in bytes. It is possible that the property size exceeds the provided buffer size. In this case, ENOMEM is returned and the buffer is filled with a partial property value.
Result	`errNone` – Succeeded. For return values other than `errNone`, see the system header file `sys_errno.h`.	

Comments	Get the specified property value from the virtual device. The return value is the error code, if any. The property is specified using a multi-character integer, such as 'size' or 'port'. Note that each property has a pre-defined size, specified in the header file for each device (such as `TwRumbler.h`). If the buffer passed in does not match the property's size, an error is returned.
Header	`TwDevice.h (included by Tapwave.h)`

TwDeviceSetProperty

Purpose	Set the specified property value for the virtual device.	
Prototype	`Err TwDeviceSetProperty (TwDeviceHandle handle, Int32 property, const void* buf, Int32 len);`	
Parameters	`[in] handle`	A `TwDeviceHandle` descriptor identifying the device for which the property value should be set.
	`[in] property`	Multi-character integer identifying the property whose value is to be set.
	`[out] buf`	Pointer to buffer containing the property's value.
	`[in] len`	Size of `buf`, in bytes.
Result	`errNone` – Succeeded. For return values other than `errNone`, see the system header file `sys_errno.h`.	

Comments	Set the specified property value for the virtual device. The return value is the error code, if any. The property is typically specified using a multi-character integer, such as 'size' or 'port'. Note that each property has a pre-defined size, specified in the header file for each device (such as `TwRumbler.h`). If the buffer passed in does not match the expected size of the property, an error is returned.
Header	`TwDevice.h (included by Tapwave.h)`

TwDeviceControl

Purpose	Send arbitrary control command to the virtual device.	
Prototype	`Err TwDeviceControl (TwDeviceHandle handle, Int32 cmd, void* buf, Int32 len);`	
Parameters	`[in] handle`	`TwDeviceHandle` descriptor identifying the device to which a control command is to be sent
	`[in] cmd`	Device-specific selector, specified as a multi-character integer
	`[in/out] buf`	Pointer to the parameter block for the device control command.
	`[in] len`	Size of `buf`, in bytes.
Result	`errNone` – Succeeded. For return values other than `errNone`, see the system header file `sys_errno.h`.	

Comments	Send an arbitrary control command to the virtual device. `cmd` is a device-specific selector. `buf` points to the command parameter block. `len` is the size of the parameter block in bytes. The parameter block may be used as input, output, or both. If the parameter block is not needed, both `buf` and `len` should be set to `0`. The return value is the error code. The `cmd` control command is specified using multi-character integers, such as 'peek', 'poll', 'push', 'pop', etc. Note that each command expects a pre-defined size for `buf`. If the buffer passed in does not match the expected size, an error is returned. The commands are defined on a device-by-device basis. See the appropriate header file for each device (such as `TwRumbler.h`) for more details.
Header	`TwDevice.h (included by Tapwave.h)`

3.6.3.Rumbler Virtual Device

The rumbler device provides a vibration effect to the user. It is implemented as an off-center weight whose rotation speed and duration can be controlled. The following describes the parameters needed to control the rumbler through the virtual device interface. See the file `TwVdRumbler.h` for more details. A higher-level wrapper routine may be provided in the future.

Note that three models are supported: the first is to turn on the rumbler, and then later turn it off. This could be used to have the device buzz while a button is held down, for example. The second model is to play a rumbler stream from a buffer. This rumbler stream is an array of UInt8 tuples, the first being a value representing the relative speed to run the rumbler at (0 meaning off), the second being a duration (in units of 1/100 second) to run the rumbler. This is used to present a one-time sensation, such as an explosion in a game. The third model is to play a rumbler stream repeatedly, until a 'stop' command is issued. This model can be used, for example, to present an ongoing sensation such as driving a vehicle over a rough road.

Even though there is only one Rumbler device on the unit, multiple opens to the rumbler device will succeed. The handles returned will have different absolute values, but they all point to a global state which is maintained internally by the system. Commands to the rumbler from multiple applications will be interleaved, with subsequent commands overriding earlier ones. i.e., "the last command wins".

3.6.3.1Sample usage:

```
#include <Tapwave.h>
// Define an array of structures containing Rumbler play
parameters:
struct TwVirtualDeviceRumbler {
     UInt8 rumbleSpeed;         // relative rotation speed, 0 =>
rumbler off.
                      // 255 => maximum speed
     UInt8 rumbleDuration; // amount of time, in 1/100 seconds,
for the
                      // device to be active
} rumbleStream [] = {{255,10}, {200,20}, {100,20}};
/* NOTE: a better way to do this, to be sure the compiler doesn't
add padding, is:
   UInt8* rumbleSteam = { '\255','\10',  '\200','\20',
'\100','\20'};  */

const void* buf = rumbleStream;
Int32 len = sizeof(rumbleStream);

TwDeviceHandle handle;

if (TwDeviceOpen (&handle, "vibrator0", "w")) {
    ErrFatalDisplayIf(1, "Missing vibrator!");
```

```
}
if (TwDeviceControl(handle, 'play', buf, len)) {
    ErrFatalDisplayIf(1, "Cannot control vibrator!");
}
if (TwDeviceClose(handle)) {
    ErrFatalDisplayIf(1, "Cannot close vibrator!");
}
```

3.6.3.2Properties:

The rumbler properties, for device `vibrator0`, can be accessed through the `TwDeviceGetProperty` and `TwDeviceSetProperty` APIs, detailed above. All these properties return a UInt8-sized value.

Property	Description
`'sped'`	The maximum speed of the rumbler device. This is the max value that can be set by the API.
`'anlg'`	A Boolean value; true if the rumbler speed is variable, false if it's binary (on/off) only.
`'dura'`	Maximum duration of rumbler ON cycle, in 1/100 of a second.
`'plng'`	A Boolean value; true if the rumbler is currently playing a stream, false if not.

Command	Description
`'strt'`	Command to turn on the rumbler. Must turn it off explicitly using `'stop'`.
`'stop'`	Command to turn off the rumbler, if playing.
`'play'`	Command to play a rumbler stream from buf. Note that an error is returned if the stream size is not a multiple of 2 (streams must contain pairs of values).
`'rept'`	Command to repeatedly play a rumbler stream from buf. Repeat until a stop command is issued, or another play or rept command is issued.
`'fast'`	Command to turn the rumbler on, fastest speed.

'medi'	Command to turn the rumbler on, medium speed.
'slow'	Command to turn the rumbler on, slow speed.

3.6.4.Audio Amplifier Virtual Device

A simple API is provided to control the muting and bass-boost capabilities of the audio system on Tapwave devices. See the file TwVdAudioAmp.h for more details.

3.6.4.1Sample Usage

```
#include <Tapwave.h>

UInt8 muteSpkr;
UInt8 buf;
Int32 len;

TwDeviceHandle handle;

if (TwDeviceOpen (&handle, "audioAmp0", "rw") != errNone) {
    ErrFatalDisplayIf(1, "Missing audio amp!");
}

len = sizeof buf;
if (TwDeviceGetProperty(handle, TW_VD_AUDIOAMP_POWER_UP, &buf,
&len") != errNone) {
    ErrFatalDisplayIf(1, "Cannot get audio amp property!");
}
if(buf == 1){
    Display(“Amp is powered on\n”);
    muteSpkr = 1;         // mute the speakers just for fun
    if (TwDeviceSetProperty(handle, TW_VD_AUDIOAMP_MUTE_SPKRS,
&muteSpkr,
                            sizeof(muteSpkr)) != errNone)
        ErrFatalDisplayIf(1, "Cannot set audio amp property!");
    else
       Display(“Speakers are now muted, enjoy the silence\n”);
}

if (TwDeviceClose(handle)) {
    ErrFatalDisplayIf(1, "Cannot close audio amp!");
}
```

3.6.4.2Properties

The Audio Amplifier properties, for device `audioAmp0`, can be accessed through the `TwDeviceGetProperty` and `TwDeviceSetProperty` APIs, detailed above. There are no read, write, or control commands associated with this device, only get and set properties.

Property	Description

`TW_VD_AUDIOAMP_ALL_PROPERTIES`	GET only, returns a bitmap of the **state of the audio amplifier** as follows:
	
	`TW_VD_AUDIOAMP_PROP_POWRD set if amplifier is on`
	`TW_VD_AUDIOAMP_PROP_MUTED set if mute-all is on`
	`TW_VD_AUDIOAMP_PROP_SPKR set if speaker mute is on`
	`TW_VD_AUDIOAMP_PROP_BASS_BOOST set if bass boost is on`

`TW_VD_AUDIOAMP_POWER_UP`	**Audio Amplifier Power**. Set to non-zero to turn on the audio amplifier system. Set to 0 to turn it off.
`TW_VD_AUDIOAMP_MUTE_ALL`	**Mute All**. Set to non-zero to mute both speakers and headphones. Set to 0 to unmute both.
`TW_VD_AUDIOAMP_MUTE_SPKRS`	**Speaker Mute**. Set to non-zero to mute the speaker only. Set to 0 to unmute the speaker only. Note that there is no way to mute the headphones.
`TW_VD_AUDIOAMP_BBOOST`	**Bass Boost**. Set to non-zero to turn bass boost on. Set to 0 to turn bass boost off.

3.7.Digital Rights Management API

DRM support is published in `TwSecurity.h`. Calling `TwSecGetFunctions` will return a function table to the DRM functions detailed below. All the TwSecurity data types, other than the function table, are opaque data structures.

Apps are signed at two levels: application signing and hardware locking, both of which are done on a secure server using Tapwave private keys and RSA algorithms. The public keys are stored in the ROM so applications and the OS can validate the signatures.

An application signature can be used to validate that the app has not been modified. The hardware locking signature can be used to validate that the application has been purchased for the device or card being used.

Verification is done by the OS when apps are launched and when certain Tapwave APIs are used. For development purposes, the API checks can be turned off by getting a Developer Access key from the Tapwave developer program. This key is in a database file called TwDevAccess and is locked to each device (so you need a separate key for each development device).

The following resources have defined uses in the DRM system (only TSIG.0 and TSIG.1 need to be created manually by the developer):

TSIG.0 – resource skip list. A list of the resources that are not included in the signing process. The format of this resource is an array of { UInt32 type; UInt16 id; UInt16 reserved; }. Developers should generate this resource list and specify any resources they plan to modify at runtime. Note that if the most significant bit (MSB) of a resource type is set to 1, then it is automatically treated as if it is in the skip list. (None of the typical 4-character resource types, e.g. 'tFRM', meet this criteria.)

TSIG.1 – require lock. 3 bytes which specify the hardware locking requirement for the app. If this resource exists then hardware locking (both device and card) is required for this app. Developers should generate this resource.

The require lock resource is of the form:

Version	1 byte – set to 1 (0x01) for this structure
Type	1 byte
LockingRequired	1 byte

Where Type is interpreted as a bitfield as follows:

0x00	none allowed.
0x01	device signature required
0x02	card signature required
0x03	allow device or card locking
0x04..0xFE	reserved for future expansion
0xFF	allow any locking type

And LockingRequired is:

0x00	locking is optional, use for "demo mode" apps
0x01	locking is required
0x02..0xFF	reserved for future expansion

TSIG.2 – application signature. The RSA signature of the application's PRC file. It includes all resources not listed in TSIG.0 and not having the most significant bit of their type set. The application signature is generated by the development server.

TSIG.3 – purchase info. A null terminated string of purchase information. The purchase info is generated by the commerce server when an application is purchased.

TSIG.4 – locking signature. A signature that locks the application to either a card or a device serial number. The locking signature is generated by the commerce server when an application is purchased.

TSIG.100 + x – code signatures. These signatures sign code.x ARM native code resources for the Tapwave Native Application Model. Only TSIG.101 is currently used to sign the sole Tapwave Native Application ARMC or ARMZ collective resource.

3.7.1.Validating the DRM system

Each app should validate it's access to the DRM system using some or all of the following methods. For additional information on validation, see the *Digital Rights Management* document.

Validate the signature of DAL.prc. DAL.prc will be signed (just like application PRCs) using the system signing key.

```
TwSecGetPublicKey (&key, twSecSystemKey);
```

- Check that the value of the DAL signing key is what it should be.
- Check that the checksum of the block of all keys is right.
- Check that the MMU setup has not been modified.
- Make sure the DRM functions point to the ROM.
- Check to make sure verification fails when it should. (This makes spoofing the verification process harder.) Do this by verifying a series of memory blocks, some of which should fail and some of which should succeed. It is computationally infeasible to know which should fail and which should succeed, so by testing a sequence of blocks you validate that DRM is functioning and hence that it's verifications of your app are likely to be working.
- Verify the signature you get for the DRM functions from `TwSecGetFunctions`.
- Make sure the DRM functions point into DRM code block.

TwSecGetFunctions

Purpose	Return a pointer to the DRM functions.
Prototype	`const TwSecTableType *TwSecGetFunctions(Int32 version, TwSecSignatureType *codeSignature, UInt8 **codeStart, UInt32 *codeSize);`

Parameters	`version`	Version of DRM functions that you want. Should be 1 for the initial version of the DRM API.
	`[out] codeSignature`	Signature returned for the code block. Indicated by `codeStart` and `codeSize`. Can be null if no value is needed.
	`[out] codeStart`	The starting address of the signed code block that contains the implementation of the DRM functions. Can be NULL if not needed..
	`[out] codeSize`	Location to store size of code that defines DRM functions. Can be null if not needed.
Result	`Returns a table of function pointers.`	
Comments	The returned table of function pointers. The pointers should all point to the code area indicated by codeStart and codeSize, which itself can be verified using the signature returned and the TAL signing key.	
Header	`TwSecurity.h`	
Sample	See DRM application in code samples.	

TwSecGetPublicKey

Purpose	Return public keys from the device ROM.	
Prototype	`Err TwSecGetPublickey(TwSecPublicKeyType *publicKey, Int32 keyNumber);`	
Parameters	`[out] publicKey`	Location to store the retrieved public key.
	`keyNumber`	Which key to retrieve. Can be any value in `[twSecFixedKeyBase, twSecFixedKeyBase - twSecNumFixedKeys – 1]`.
Result	Error code on failure: `sysErrNotAllowed` – if keys have been modified.	
Comments	Using the constant, `twSecSystemKey`, will return the System Signing Key which can also be retrieved directly from the EEPROM.	
Header	`TwSecurity.h`	
Constants	`twSecSystemKey`	
Sample	See DRM application in code samples.	

TwSecGetHardwareId

Purpose	Return the hardware id of a specific hardware item.
Prototype	`Err TwSecGetHardwareId(TwSecHardwareIdType *hardwareId, UInt8 hardwareInfoType, Int16 slotNum);`

Parameters	`[out] hardwareId`	Location to store hardware id.
	`HardwareInfoType`	Which type of hardware id to get. See constants below.
	`SlotNum`	If getting card id, which slot to get info for.
Result	*`Returns a non-zero error code on failure. sysErrParamErr or Expansion manager errors.`*	

Comments	The constants that are accepted for `hardwareInfoType` are
	`twSecDevice`
	and
	`twSecCard`	

Header	`TwSecurity.h`	
Constants	See above	
Sample	See DRM application in code samples.	

TwSecVerifyDatabase

Purpose	Do a full verification on a database. Both application signing and hardware locking are checked. Databases without signatures are defined to be invalid.

Prototype	`ARM:`
	`Boolean TwSecVerifyDatabase(MemHandle dbH, UInt32 *failureReason, UInt32 failureAction);`
	`68k:`
	`Boolean TwSecVerifyDatabase(UInt16 cardNo, LocalID dbId, UInt32 *failureReason, UInt32 failureAction);`

Parameters	ARM: dbH 68k: cardNo dbId	Identification of which database to check: ARM: Handle to the database. 68k: Card number and LocalID of database.
	[in/out] failureReason	Error code indicating reason for failure. On success, the passed in value is rotated by 19 bits (see comments).
	failureAction	Indicates what actions the function should take on failure. The choices are return error code or reset (see constants below).
Result	Returns true if database is valid.	
Comments	Upon successful verification, the failureReason value is rotated by 19 bits so that it is hard to trick the application into calling a fake entry point in ROM that just returns true (fake entry points are unlikely to also rotate the value in failureReason by 19 bits!). If the database is open for write access, verify will fail because read access is needed to verify.	
Header	TwSecurity.h	
Constants	Failure reasons will be one of the following twResetReasonInvalidAppSig twResetReasonInvalidHwrSig The failureAction codes are twSecReturnOnFail twSecResetOnFail	

TwSecVerifyCurrentApp

Purpose	Do the most complete job possible, from the current execution point, to verify that the app running is authorized to run.	
Prototype	`Boolean TwSecVerifyCurrentApp(UInt32 *failureReason, UInt32 failureAction);`	
Parameters	`[in/out] failureReason`	Error code indicating reason for failure. On success, the passed in value is rotated by 19 bits (see comments).
	`failureAction`	Indicates what actions the function should take on failure. The choices are return error code or reset (see constants below).
Result	True if verification succeeded.	
Comments	Upon successful verification the `failureReason` value is rotated by 19 bits so that it is hard to trick the application into calling a fake entry point in ROM that just returns true (fake entry points are unlikely to also rotate the value in `failureReason` by 19 bits!).	
Header	`TwSecurity.h`	

Constants	`The failureAction codes are`
	`twSecReturnOnFail`
	`twSecResetOnFail`	

Sample	See DRM application in code samples.	

TwSecFailureReset

Purpose	Reset the device in such a way that the reason for the reset displays after the reset.
Prototype	`void TwSecFailureReset(const char *filename, UInt32 reason, const char *message);`

Parameters	[in] filename	Name of file where error occurred.
	reason	One of the constants below.
	[in] message	Message to display.
Result	nothing	
Header	TwSecurity.h	
Constants	Reason code can be one of the following: twResetReasonInvalidAppSig twResetReasonInvalidHwrSig twResetReasonInvalidCodeSig twResetReasonInvalidDevSig	

TwSecVerifyMemory

Purpose	Verify a block of memory.	
Prototype	Boolean TwSecVerifyMemory(const UInt8 *mem, UInt32 memSize, const TwSecPublicKeyType *key, const TwSecSignatureType *sig, UInt32 *failureReason, UInt32 failureAction);	
Parameters	[in] mem	Starting address of memory block to verify.
	memSize	Number of bytes to verify.
	[in] key	Public key to use in verify.

	[in] sig	Signature of memory block.
	[in/out] failureReason	Error code indicating reason for failure. On success, the passed in value is rotated by 19 bits (see comments).
	failureAction	Actions the function should take on failure. The choices are return error code or reset (see constants below).
Result	Returns true if verified.	
Header	TwSecurity.h	
Constants	The failureAction codes are twSecReturnOnFail twSecResetOnFail	
Sample	See DRM application in code samples.	

TwSecVerifyPointerOwnership

Purpose	Verify that the database has a valid signature and that the pointer points to data that belongs to the database.	
Prototype	ARM: Boolean TwSecVerifyPointerOwnership(UInt16 cardNo, LocalID dbID, const TwSecPublicKeyType *key, const void *ptr, UInt32 *failureReasonP, UInt32 failureAction); 68K: Boolean TwSecVerifyPointerOwnership(MemHandle dbH, const TwSecPublicKeyType *key, const void *ptr, UInt32 *failureReasonP, UInt32 failureAction)	
Parameters	ARM: dbH 68k: cardNo	Identification of which database to check: ARM: Handle to the database.

	dbID	68k: Card number and LocalID of database.
	[in] key	Public key to use in verify. You can pass NULL to use the default system key for the application.
	[in] ptr	Pointer to verify.
	[in/out] failureReason	Error code indicating reason for failure. On success, the passed in value is rotated by 19 bits (see comments).
	failureAction	Actions the function should take on failure. The choices are return error code or reset (see constants below).
Result	Returns true if verified, returns false and error code or resets if verification fails.	
Header	TwSecurity.h	
Constants	The failureAction codes are twSecReturnOnFail twSecResetOnFail	
Sample	See DRM application in code samples.	

3.8.Desktop APIs

In order to provide support for installing data files, Tapwave has defined a new Install Aide function. This function is defined in `InstAide.dll`.

PlmSlotInstallFileToDir

Purpose	Install a file to a specific directory.

Prototype	`int WINAPI PlmSlotInstallFileToDir (DWORD dwUserId, DWORD dwSlotId, const TCHAR *pszFilePath, const TCHAR *pszDir, DWORD dwCondId);`	
Parameters	`[in] dwUserId`	The user's Palm ID as returned by `PlmGetUserIDFromName`.
	`[in] dwSlotId`	The ID number of the slot to install into. The slot can be retrieved using `UmSlotGetInfo`. You'll usually use slot 0 which is the internal VFS volume.
	`[in] *pszFilePath`	The source file to install. This must include the full path.
	`[in] *pszDir`	The destination directory on the card, starting at the root level. (e.g: "PALM\Programs\<gamename>-<gamecreatorid>". This should not start or end with a backslash character.
	`[in] dwCondId`	The creator ID of the Tapwave card installer. Use `'TWci'`.
Result	`ERROR_SUCCESS` – Successful installation. `ERR_PILOT_INVALID_FILENAME` - if `pszFilePath` is NULL `ERR_PILOT_INVALID_FILE_TYPE` or `ERR_PILOT_INVALID_PATH` - if there was a problem retrieving the Install path. `ERR_PILOT_FILE_ALREADY_EXISTS` - if file already exists in the user's slot install directory. `ERR_PILOT_INVALID_SOURCE_FILE` - if the source file could not be located. `ERR_PILOT_COPY_FAILED` - if an error occured during the actual copy operation.	

Comments	This function sets the appropriate registry flags to schedule the conduit, creates the install directories under the user folder on the desktop, and copies the file to the correct location to await installation by the new Tapwave install conduit.