Macintosh Developer's Guide to the Wacom Tablet

The purpose of this document is to help developers write tablet aware applications for the Apple MacOS X operating system.

This document explains how to get and interpret tablet data from either a Carbon CFM, Carbon Mach-O, or Cocoa/Appkit application. It will also provide many tips and tricks, such as how to create a tablet aware application that works in both MacOS X and the Classic MacOS.

Audience

This document is intended for software engineers adding tablet capabilities to MacOS X applications using either the Carbon or Cocoa frameworks.

Carbon: This document is intended for MacOS X developers who are at least familiar with the Carbon API set. The developer should have an understanding of event handlers and be able to install and use them. For more information on Carbon event handlers, please refer to Apple's Inside Carbon: Carbon Event Manager.

Cocoa: Cocoa developers should have a basic understanding of the AppKit, NSNotifications and Categories. Advanced tablet usage requires a basic understanding of Carbon events and how to access them. Most of this document is written from the Carbon perspective. Cocoa specific notes will be labeled as such.

Glossary

AppKit – A developer framework built using the Cocoa API and Objective-C language.

CFM – Code Fragment Manager. CFM is the native run time environment for Classic MacOS. CFM applications that strictly rely on the Carbon API can also run natively on MacOS X.

Carbon – A MacOS X API set that is based on the Classic MacOS API. Programs written using the Carbon API generally can be compiled to run natively on both MacOS X, and the Classic MacOS. One major notable exception is that all of the Tablet functionality included in the Carbon API only work properly in MacOS X.

Cocoa – An API set based on the NextStep API. Cocoa applications are usually written using the AppKit framework.

Mach-O – The native run time environment for MacOS X. Mach-O applications can use either the Cocoa or Carbon API set and will only run on MacOS X. Though CFM is also a native run time environment for MacOS X, special care must be taken when trying to get Mach-O applications and libraries to communicate with CFM applications and libraries.

SDK – Software Developers Kit. Sample code projects to help third party developers write tablet aware applications.

Transducer – A device that works with the tablet. Ex: Pen, Puck (Mouse), Stroke Pen, Airbrush, 4D Mouse.

General

Tablet Theory

Tablets provide an application with more accurate and more pieces of data than a mouse can. Perhaps the biggest tablet feature an application can take advantage of is pressure. Stylus like devices that are placed on a tablet will report how hard the user is pressing the device against the tablet. With just this small piece of data, a user can tell an application to vary the thickness of a line being drawn, or the opacity, or color. Perhaps an application can determine how fast to scroll, or how loud to play a note. Wacom tablets can also provide even more pieces of data such as the location of the device between pixels, the tilt of a stylus, the rotation of a mouse and more.

There are many different types of devices with different features that can be used on a tablet. Some Wacom tablets can even support having two devices on the tablet at the same time. To help handle all of these different devices, a special type of event is introduced, a proximity event. When a user places an appropriate device close enough to a tablet (not necessarily touching the tablets) to start sending information, the tablet sends a proximity event announcing that a tool has entered proximity. Also when the user moves a device too far away from the tablet, a proximity event is sent announcing that the tool has left proximity. An application should use these events to determine the current tool's abilities, ID, and type. Then, as the application gets data from the tool, it will know how to use it.

For example, in a painting application, if the proximity event informs the tablet that the tip of a tool is being used, the application should draw lines. However, if the proximity event informs the tablet that the erasure end of a tool is being used, then the application should erase instead of draw.

Tablet Support In MacOS X

Tablets are full-fledged citizens in MacOS X with their own events defined in CarbonEvents.h. This makes it easy to implement most of the features of the tablet without having to worry about opening / attaching to a driver. Normally, these tablet events are embedded inside of mouse events. Thus, all you have to do is to check if the mouse event has tablet data associated with it. If so, use it. If not, then the user is using a mouse and not a tablet device.

Cocoa:
Some of the tablet's features have been added directly to the Mouse events in AppKit. Most of them have not. However, with just a little carbon know how (which we will explain) you can get the Carbon Tablet data from Cocoa/Appkit.

Tablet Events

There are two types of tablet events in MacOS X, proximity and pointer.

Proximity events describe what kind of device is being used and are issued whenever a transducer is placed near the tablet, or removed from the tablet. Devices that are in proximity can cause pointer events to occur. Proximity events are important because they tell you what kind of device is being used on a tablet, what the capabilities of that device are, and provides key information that is used to link pointer events with specific proximity events.

Note: You must use the proximity event to determine if the tip or erasure end of a pen is in use.

Pointer events describe the current state of a transducer that is in proximity and are issued whenever any changes are made to the state of a transducer. Pointer events contain information like the location of the transducer, the pressure and tilt of the device.

Tablet Data Structures

Proximity Event

Related Data Structures, Enums, & Constants

```
kEventParamTabletProximityRec        defined in CarbonEvents.h
  typeTabletProximityRec               defined in CarbonEvents.h

  struct TabletProximityRec {          defined in CarbonEvents.h
       UInt16 vendorID;
       UInt16 tabletID;
       UInt16 pointerID;
        UInt16 deviceID;
       UInt16 systemTabletID;
       UInt16 vendorPointerType;
       UInt32 pointerSerialNumber;
       UInt64 uniqueID;
       UInt32 capabilityMask;
       UInt8 pointerType;
       UInt8 enterProximity;
  };
  typedef struct TabletProximityRec TabletProximityRec;
```

vendorID

Type: UInt16

Notes:
This is the USB Vendor ID of the tablet. For Wacom, the vendor ID is 0x56A. You can generally ignore this field unless you want to decode the vendorPointerType.

tabletID

Type: UInt16

Notes:
This is the USB Model ID of the tablet. You can generally ignore this field.

pointerID

Type: UInt16

Notes:
This is the index of the device on the tablet. Usually this is 0, but for tablets that support multiple concurrent devices (also known as multimode or Dual Tracking), this number will change for the second concurrent device and so on. If your app does not support multiple concurrent devices and the pointerID is not 0, then you may want to make a note of the deviceID in this event and ignore tablet pointer events with this deviceID until you get a proximity event with this deviceID and a pointerID of 0.

If your application does support Dual Tracking then it is important to note that only one device should move the cursor. See Advanced Techniques . Dual Track for more information.

deviceID

Type: UInt16

Notes:
This is a Special ID that is used to match pointer events with this proximity event. It is the applications responsibility to save any relevant information provided in the proximity event structure for future reference while this device is in proximity. All of the pointer events produced by this device between an enter and leaving proximity events will contain the same deviceID as the one here.

systemTabletID

Type: UInt16

Notes:
This is the index of the tablet on the system. If there are multiple tablets connected to a system, this field will tell you which tablet this device is being used on.

vendorPointerType

Type: UInt16

Notes:
This is a coded bit field that you can use to determine exactly what type of device is being used. Some bits have special meaning to the hardware, but are not of interest to a developer. These bits are Wacom specific, so you should verify that the vendorID is Wacom's.

What is of interest is:
There are 12 bits total. The upper 4 bits, and a couple of bits in the lower four, identify the device. The middle 4 bits are used to differentiate between electrically similar, but physically different devices, such as a general stylus with a different color. So to figure out a specific device type you mask with 0x0F06. For example maybe 0x0812 is a stylus that is black, 0x0802 is the standard stylus included in the box, and 0x0832 may be a stylus with no side switches.

The currently supported types are:

General Stylus:	(vendorPointerType & 0x0F06) == 0x0802
Airbrush:	(vendorPointerType & 0x0F06) == 0x0902
General Mouse:	(vendorPointerType & 0x0F06) == 0x0006
Pro Mouse:	(vendorPointerType & 0x0F06) == 0x0004
Rotation Stylus:	(vendorPointerType & 0x0F06) == 0x0804

Intuos

Standard Stylus	= 0x822
Inking Stylus	= 0x812
Stroke Stylus	= 0x832
Grip Stylus	= 0x842
Airbrush	= 0x912
4D Mouse	= 0x094
Lens Cursor	= 0x096

Intuos 2

Standard (Grip) Stylus	= 0x852
Classic Stylus	= 0x822

Inking Stylus	= 0x812
Stroke Stylus	= 0x832
Designer Stylus	= 0x842
Airbrush	= 0x912
2D Mouse	= 0x007
4D Mouse	= 0x094
Lens Cursor	= 0x096

Intous 3

Standard (Grip) Stylus	= 0x823
Rotation Stylus (Art Pen)	= 0x885
Inking Stylus	= 0x801
Airbrush	= 0x913
2 D mouse	= 0x017
Lens Cursor	= 0x097

Graphire & Graphire 2 & Graphire 3 & Bamboo

Graphire Stylus	= 0x022
Graphire Mouse	= 0x296

pointerSerialNumber

Type: UInt32

Notes:
Serial number of the transducer. Each transducer has a unique number within each device type. Therefore it is possible for a puck and a pen to have the name serial number.

Wacom's consumer level line of tablets, such as the Graphire, do not support this.

uniqueID

Type: UInt64

Notes:
Also known as Tool ID. A chip with a unique number is inside every device so every device can be uniquely identified. With Unique ID you can assign a specific tool to a specific device or use it to "sign" a document. You can restrict access to document layers to particular devices and have settings follow a device to other machines.

Wacom's consumer level line of tablets, such as the Graphire, do not support this, however, different IDs from such a tablet would signify that a different type of device is being used, so you could customize all the pens of a Graphire tablet one way, and the mice another way

Important:
In the Classic MacOS & PC Tablet documentation, Tool ID is often referred to as Device ID. It is important to remember that the MacOS X proxEvent.deviceID is not the same as Tool ID. It is the uniqueID field that should be used to uniquely identify a device.

capabilityMask

Type: UInt32

Notes:
This is a Bit field used to determine what capabilities this device has. Use the following masks that are defined in Wacom.h on this field to help determine what the capabilities of this device are.

kTransducerDeviceIdBitMask	This bit is set to indicate that the deviceID field of a pointer record will contain valid data. Considering that deviceID is used to link pointer events with proximity events, if a device does not support this, then something is terribly wrong.
kTransducerAbsXBitMask	This bit is set to indicate that the absX field of pointer records with this deviceID contain valid data.
kTransducerAbsYBitMask	This bit is set to indicate that the absY field of pointer records with this

	deviceID contain valid data.
kTransducerVendor1BitMask	This bit is set to indicate that the vendor1 field of pointer records with this deviceID contain valid data.
kTransducerVendor2BitMask	This bit is set to indicate that the vendor2 field of pointer records with this deviceID contain valid data.
kTransducerVendor3BitMask	This bit is set to indicate that the vendor3 field of pointer records with this deviceID contain valid data.
kTransducerButtonsBitMask	This bit is set to indicate that the buttons field of pointer records with this deviceID contain valid data.
kTransducerTiltXBitMask	This bit is set to indicate that the tiltX field of pointer records with this deviceID contain valid data.
kTransducerTiltYBitMask	This bit is set to indicate that the tiltY field of pointer records with this deviceID contain valid data.
kTransducerAbsZBitMask	This bit is set to indicate that the absZ field of pointer records with this deviceID contain valid data..
kTransducerPressureBitMask	This bit is set to indicate that the pressure field of pointer records with this deviceID contain valid data.
kTransducerTangentialPressureBitMask	This bit is set to indicate that the tangentialPressure field of pointer records with this deviceID contain valid data.
kTransducerOrientInfoBitMask	Deprecated! Don't Use!
kTransducerRotationBitMask	This bit is set to indicate that the rotation field of pointer records with this deviceID contain valid data.

pointerType

Type: UInt8

Notes:
This is the generic type of device.. Pen, Puck or Eraser. The following code snippit is defined in Wacom.h. Compare these enums with the pointerType.

```
typedef enum EPointerType
{
EUnknown = 0,     // should never happen
EPen,             // tip end of a stylus like device
ECursor,  // any puck like device
EEraser           // eraser end of a stylus like device
} EPointerType;
```

enterProximity

Type: UInt8

Notes:
Is the device entering proximity? This field indicates whether the transducer is being put near the tablet, or taken away. non-zero = entering; zero = leaving

Pointer Event

Related Data Structures, Enums, & Constants

`KEventParamTabletPointRec`	defined in CarbonEvents.h
`TypeTabletPointRec`	defined in CarbonEvents.h
`struct TabletPointRec {`	defined in CarbonEvents.h

```
        struct TabletPointRec {
             SInt32 absX;
             SInt32 absY;
              SInt32 absZ;
             UInt16 buttons;
             UInt16 pressure;
             SInt16 tiltX;
        SInt16 tiltY;
        UInt16 rotation;
        SInt16 tangentialPressure;
        UInt16 deviceID;
        SInt16 vendor1;
        SInt16 vendor2;
        SInt16 vendor3;
    };
    typedef struct TabletPointRec TabletPointRec;
```

absX

Type: SInt32
Range: 0 to tablet width in counts. The width counts depend on model and size of tablet.

Notes:
This is the X coordinate of the device on the tablet.
If you really want to do the scaling yourself, then use this value (see Advanced Techniques . Scaling), otherwise use the sub-pixel information from the mouse event by getting the kEventParamMouseLocation of typeHIPoint (see Advanced Techniques . Sub Pixels).

absY

Type: SInt32
Range: 0 to tablet height in counts. The height counts depend on model and size of tablet.

Notes:
This is the Y coordinate of the device on the tablet.
If you really want to do the scaling yourself, then use this value (see Advanced Techniques . Scaling), otherwise use the sub-pixel information from the mouse event by getting the kEventParamMouseLocation of typeHIPoint (see Advanced Techniques . Sub Pixels).

absZ
Type: SInt32
Range: -1023 to 1023

Notes:
This is the Z coordinate of the device on the tablet. This is NOT pressure!
Wacom's Intuos and Intuos 2 4D Mice have a wheel that can report the Z coordinate. A possible use of the 4D Mouse wheel is for zooming or for 3D navigation. Note that the wheel must be set to "Application Defined" in the Wacom Tablet Control Panel for the absZ field to update. All other devices and tablets do not support Z.

buttons

Type: UInt16
Range: N/A

Notes:
This is a Bit field used to determine the state of the buttons on a transducer. Bit 0 is the first button. A 1 means the button is being pressed. For a stylus, the tip is considered to be the first button. All Wacom tablets and devices support buttons.

pressure

Type: UInt16
Range: 0 to (2^16)-1

Notes:
This is the scaled pressure that is applied to a transducer.

This is probably the most important field. All stylus like transducers and even some mice transducers have pressure. If a transducer is in proximity, but not actually touching the tablet, then the pressure is 0. As the user presses a stylus tip, or eraser, to the tablet, this value will change according the amount of pressure they apply to the stylus. All Wacom tablets support pressure, however, not all devices support pressure!

tiltX

Type: SInt16
Range: -((2^15)-1) to (2^15)-1 or (-32767 to 32767)

Notes:
This is scaled tilt amount of a transducer in the X direction. If a stylus like device is held perfectly perpendicular to the tablet, then the tilt value is 0. As the stylus is tilted to the left, the value becomes more and more negative until the pen is parallel to the tablet (-90°).. As the stylus is tilted to the right, the values become more and more positive until the pen is parallel to the tablet (-90°)..

tiltY

Type: SInt16
Range: -((2^15)-1) to (2^15)-1 or (-32767 to 32767)

Notes:
This is scaled tilt amount of a transducer in the Y direction. If a stylus like device is held perfectly perpendicular to the tablet, then the tilt value is 0. As the stylus is tilted to the top, the value becomes more and more negative until the pen is parallel to the tablet (-90°). As the stylus is tilted to the bottom, the values become more and more positive until the pen is parallel to the tablet (-90°).

Tip: Convert To Polar Coordinates

You can easily convert tiltX and tiltY to polar coordinates with the following code:

```
void ConvertCartesianToPolar(double xCoord, double yCoord,
double *r, double *theta)
{
if ( r != NULL)
{
*r = sqrt( (xCoord * xCoord) + (yCoord * yCoord) );
}

if ( theta != NULL )
{
*theta = atan2( yCoord, xCoord );
}
}
```

Not all Wacom Tablets support tilt. On tablets that do support tilt, only stylus like devices (including the airbrush) have tilt.

rotation

Type: UInt16
Range: 0 to 23040

Notes:
This is a fixed-point (10.6 format.) representation of device rotation in degrees. That is, the first 10 bits are the integer part, and the last 6 bits are the fractional component. Therefore, to convert to floating point notation, divide the fixed-point rotational value by 2^6 (or 64).

rotDegrees = (float)rotation/64.0;
rotRadians = (rotDegrees * PI) / 180.0;

Note: Rotation is the angle between the "front" of the transducer and the top of the tablet. This does not mean much for mice that support rotation, however, it is very important for styli. For styli this rotation is different than "twisting" the barrel of the pen. While twisting the barrel of the pen does cause a rotation change, the rotation can change without actually twisting the barrel of the pen. For example, hold the pen upright with side-switch buttons pointed towards the top of the tablet. Now fully tilt the pen to the left. Rotate the pen so that the end is pointing down. Finally, tilt the pen back upright. Notice how the side switches are rotated 90° from where they started and that you never physically twisted the barrel of transducer.

tangentialPressure

Type: SInt16
Range: -((2^15)-1) to (2^15)-1 or (-32767 to 32767)

Notes:
This is another pressure field a device can have. It is also often referred to as barrel pressure. The wheel on the Intuos and Intuos 2 Airbrush tool are the only tools that report tangential pressure.

Wacom interprets the tangential pressure as follows. 0 is the neutral position. Movement away from the neutral position can then be either positive or negative. The Airbrush's tangential pressure wheel, however, can only move in the positive direction from the neutral position. (Fingerwheel all the way to the front is neutral.)

Tip: Tangential Effect

You may want to scale the tangential pressure to get full effect at full pressure.

```
#define      kTanPressureLevels          32767

// This goes in your draw routine
// Do not forget to check the capabilities mask from the enter proximity
// event of this deviceID to make sure that this device has tangential
// pressure.

tangentialPressure = pointerEvent.tangentialPressure;

tangentialEffect = ( tangentialPressure * 100% ) / kTanPressureLevels;
```

deviceID

Type: UInt16
Range: N/A

Notes:
This is a Special ID that is used to match this pointer event with a previous proximity event. Refer to the data stored from the associated proximity event to determine the capabilities of this device and to determine if it is a tip, erasure or mouse.

vendor1

Type: SInt16
Range: N/A

Notes:
Reserved.

vendor2

Type: SInt16
Range: N/A

Notes:
Reserved.

Tablet Data Structures . Pointer Event . vendor3

Type: SInt16
Range: N/A

Notes:
Reserved.

Tablet Events

Tablet events may get sent to an application in two slightly different ways. Generally, tablet events are embedded inside mouse events because the tablet is also used to control the screen cursor. In some cases, where there is no cursor movement, an application may receive a tablet event by itself. Such tablet events are referred to as "pure tablet events."

The best way to get tablet events is to install Carbon Event Handlers for mouse events and for the tablet events. Typically, applications only care about Drag, Up and Down events. You can ignore Mouse Move events unless your application cares about mouse or tablet data when the user is not starting, in the middle of, or ending a drag. There is a very special case, though, right after a mouse event that you should be aware of. To prevent accidental drags when the user is attempting to click, the Wacom driver does not produce mouse drag events until the user has moved at least one pixel. That means that there may be pressure changes and other tablet data that does not come through embedded in a mouse event. For many applications, this is not a problem. However, if this is a problem for your app, the tablet data is being supplied as pure tablet pointer events. If you listen for and get pure tablet pointer events right after a mouse down, consider the mouse location to be the same and only the tablet data has changed. Once the tablet driver starts outputting drag events, it will continue to do so until all the buttons and the pen tip is released. It should also be noted that during a drag, the tablet is outputting a constant stream of events. This means that you may receive events that are exactly like a previous tablet event. The user must have a very steady hand to do this, but it is not uncommon for this to occur either at full pressure or for short intervals.

Proximity events are always sent as pure tablet events. This way you don't have to search each mouse event to get the proximity events. Simply install a carbon tablet class, tablet proximity event handler and you will get all proximity events without digging for them. Also, you should install an event handler on the "Monitor Target" in 10.3.x or higher. This will allow you to get Proximity events while your application is in the background. This is important, as the user can change tools, or flip over the pen to the eraser while your app is in the background. Without a handler installed on the monitor event target, you will not be informed of this. Proximity events may also be posted embedded in mouse events. You can ignore these proximity events, but you can not assume that embedded tablet events are always pointer events. Always check the embedded event type in mouse events!

The Carbon mouse and tablet event types are defined in CarbonEvents.h. If you do not know how to install Carbon Event Handlers than please refer to Apple's online documentation, Inside Carbon: Carbon Event Manager.

Extracting from Mouse Events

Once you receive a mouse event, you can tell if a tablet has generated it by asking for the `kEventParamTabletEventType` event parameter. If this parameter exists then it is safe to assume tablet data exists. Based on the event type, you can then get the appropriate tablet data by passing either `kEventParamTabletProximityRec` or `kEventParamTabletPointerRec` to `GetEventParameter()`. The following sample code demonstrates how to extract the embedded tablet data.

```
OSStatus MyMouseEventHandler( EventHandlerCallRef inCallRef, EventRef inEvent,
                                  void* userData )
{
UInt32            eventType;
OSStatus  result;

result = GetEventParameter( inEvent,
                           kEventParamTabletEventType,
                             typeUInt32,
                           NULL,
                             sizeof( eventType ),
                           NULL,
                             &eventType );
if ( result != noErr )
{
       // not generated with a tablet
       // Treat this event like it came from a standard mouse.
return eventNotHandledErr;
}

switch ( eventType )
{
case kEventTabletProximity:
{
TabletProximityRec proxEvent;

result = GetEventParameter(inEvent,
                                    kEventParamTabletProximityRec,
                                    typeTabletProximityRec,
                                    NULL,
                                    sizeof( TabletProximityRec ),
                                        NULL,
                                        &proxEvent);

                if ( result == noErr )
                {
                // proxEvent now contains the embedded tablet proximity event
                // Do something with it here.
                }
                else
                {
                // Oh No!
                // Bad result from GetEventParameter(), getting
// kEventParamTabletProximityRec
                assert(0);
                }
                break;

                case kEventTabletPoint:
                {
                TabletPointerRec pointerEvent;

                result = GetEventParameter(inEvent,
                                           kEventParamTabletPointerRec,
                                           typeTabletPointerRec,
                                           NULL,
                                           sizeof( TabletPointerRec ),
                                           NULL,
                                           &pointerEvent);

                if ( result == noErr )
                {
                // pointerEvent now contains the embedded tablet
                // pointer event
                        // Do Something with it here.
                }
                else
                {
                // Oh No!
                // Bad result from GetEventParameter(), getting
// kEventParamTabletPointerRec
                         assert(0);
                }
                }
                break;

                default:
                {
                // Oh No! How'd this happen
                // Unknown event type

                assert( 0 );
                }
                break;
        }

        return noErr;
}
```

Tip:
A really easy way to handle embedded proximity events is just to ignore them. It will be duplicate information from the pure tablet proximity event. You do have a Carbon event handler for pure tablet proximity events right?

Caution:
Do not depend on WaitNextEvent() to return mouse event records. Old style event records do not contain embedded tablet information. You must use Carbon Event Refs. If you must, replace WNE with RecieveNextEvent(). ReceiveNext event will return a Carbon Event Ref, and you can convert that to an old style Event Record if you need to for further processing.

Tracking Loops

When your application receives a mouse down event, you will probably want to go into some sort of tracking loop until your application receives a mouse up event. You must be careful how you write your tracking loop or you will find that your application's mouse and tablet event handlers do not get called.

Due to the symmetric multi-tasking of MacOS X, Apple suggests that you no longer use the old event routines like StillDown. To this end, Apple has added two new mouse tracking routines, TrackMouseLocation() and TrackMouseLocationWithOptions(). These two new API calls do not return the mouse event, nor do they call your mouse or tablet event handlers. The embedded tablet events get eaten by these calls, so do NOT use TrackMouseLocation() or TrackMouseLocationWithOptions().

***** Do NOT use TrackMouseLocation() or TrackMouseLocationWithOptions() *****

The solution is to use the new `RecieveNextEvent()` and roll your own `TrackMouseLocation`. There are 3 important things that `ReceiveNextEvent()` (RNE) does for Tablet Aware application developers. First is that RNE blocks for the specified time if there are no events to get. Second, while RNE is blocked, any timers you may have running still fire. And finally, RNE returns the event as a Carbon EventRef so that you can get to all the mouse information and the embedded tablet event.

Sample Code: WacomTrackMouseReturnEvent

```
///////////////////////////////////////////////////////////////////////////////
/*
*     WacomTrackMouseReturnEvent
*
* Once entered this function waits for a mouse event or tablet event. When
* one of these events occur, the function returns the EventRef containing
* the mouse or tablet event to the caller. It is the caller's responsibility
* to dispose of the event. While there is no activity, the current event loop
* is run, effectively blocking the current thread (save for any timers that
* fire). This helps to minimize CPU usage when there is nothing going on.
*
* IMPORTANT:
* It is the caller's responsibility to dispose of the returned event using
* ReleaseEvent( eventRef );
*/
OSStatus WacomTrackMouseReturnEvent(EventRef *eventRef)
{
const EventTypeSpec     events[ 6 ] = {
{ kEventClassMouse, kEventMouseDown },
{ kEventClassMouse, kEventMouseUp },
{ kEventClassMouse, kEventMouseMoved },
{ kEventClassMouse, kEventMouseDragged },
{ kEventClassTablet, kEventTabletPointer },
{ kEventClassTablet, kEventTabletProximity } };
OSStatus         result = noErr;

// Check make sure eventRef exists. Avoid crashing.
if(eventRef == NULL)
{
return paramErr;
}
// Look for tablet & mouse events.
result = ReceiveNextEvent( 6,
events,
kEventDurationForever,
true,
eventRef );
return result;
}
```

Sample Code: WacomTrackMouseLocationUsingHandlers

```
typedef UInt16 WacomTrackingResult;
enum {
kWacomTrackingMouseDown = 1,
kWacomTrackingMouseUp = 2,
kWacomTrackingMouseExited = 3,
kWacomTrackingMouseEntered = 4,
kWacomTrackingMouseDragged = 5,
kWacomTrackingKeyModifiersChanged = 6,
kWacomTrackingUserCancelled = 7,
kWacomTrackingTimedOut = 8,
kWacomPureTabletEvent = 9,
};

///////////////////////////////////////////////////////////////////////////////
/*
* WacomTrackMouseLocationUsingHandlers
*
* This function is similar to Apple's TrackMouseLocation. The difference is
* that this function will call your Carbon Event Handlers for the mouse and
* tablet events. Also, the mouse location is returned in a HIPoint that is
* local to inPort as opposed to a Point (typeQDPoint).
*
* inPort can be NULL, if so the current port is used.
*/
OSStatus WacomTrackMouseLocationUsingHandlers(GrafPtr inPort,
HIPoint * outPt,
WacomTrackingResult * outResult)
{
// List of events we want to track
const EventTypeSpec     events[ 5 ] = {
{ kEventClassMouse, kEventMouseDown },
{ kEventClassMouse, kEventMouseUp },
{ kEventClassMouse, kEventMouseDragged },
{ kEventClassTablet, kEventTabletPointer },
{ kEventClassTablet, kEventTabletProximity } };

// Function Variables
EventRef  eventRef;
GrafPtr          oldPort;
GrafPtr          curPort;
HIPoint          locHIPoint;
HIPoint          deltaHIPoint;
Point            aQDPoint;
Rect             portRect;
```

```
OSStatus  result = noErr;

*outResult = kWacomTrackingTimedOut;

// Look for tablet & mouse events.
result = ReceiveNextEvent( 5,
events,
kEventDurationForever,
true,
&eventRef );
if ( result == noErr )
{
// Initially guess at result type.
if (GetEventClass(eventRef) == kEventClassMouse)
{
switch (GetEventKind(eventRef))
{
case kEventMouseDown:
*outResult = kWacomTrackingMouseDown;
break;

case kEventMouseUp:
*outResult = kWacomTrackingMouseUp;
break;

case kEventMouseDragged:
*outResult = kWacomTrackingMouseDragged;
break;

default:
break;
}
}
else
{
// If this is not a mouse event then it must be a tablet event
// in which case the outPt will not be valid.
*outResult = kWacomPureTabletEvent;
}

// Make sure outPt is not NULL or else a crash will occur
if (outPt != NULL)
{
GetPort(&oldPort);

// Set the current grafPort to inPort. If inPort == NULL then use
// the current grafPort.
if ( inPort != NULL )
{
SetPort(inPort);
curPort = inPort;
}
else
{
curPort = oldPort;
```

```
}

// Get the mouse location of the event as a HIPoint, so that we
// get sub-pixel information. If this is not a mouse event (ie
// tablet event) then GetEventParameter will return an error and
// we ignore trying to return the mouse loc.
if(noErr == GetEventParameter(eventRef,
kEventParamMouseLocation,
typeHIPoint, NULL,
sizeof(HIPoint), NULL,
(void *)&locHIPoint))
{
// Must convert the point from Global to Local coordinates
// but we must be careful and maintain the sub-pixel information.
aQDPoint.h = locHIPoint.x;
aQDPoint.v = locHIPoint.y;

GlobalToLocal(&aQDPoint);

locHIPoint.x = (locHIPoint.x - trunc(locHIPoint.x)) +
(double)aQDPoint.h;
locHIPoint.y = (locHIPoint.y - trunc(locHIPoint.y)) +
(double)aQDPoint.v;

// return the local mouse location, however, if outPt is NULL,
// we do not want to crash. But, we still need the current mouse
// location to determine if this really should be called an enter
// or exit result.
if ( outPt != NULL )
{
*outPt = locHIPoint;
}

// If we can get a mouse delta then we can determine the mouse
// location of the mouse event before this one and can thus check
// if this is an enter or exit or plain drag event.
// GetEventParameter will return an error if this is not a mouse
// move or drag. But in such a case, the track result should not
// be Enter or Exit and was thus handled above properly.
if(noErr == GetEventParameter(eventRef,
kEventParamMouseDelta,
typeHIPoint, NULL,
sizeof(HIPoint), NULL,
(void *)&deltaHIPoint))
{
Boolean oldPointInBounds;
Boolean newPointInBounds;

GetPortBounds(curPort, &portRect);

// Check to see if the mouse loc for this event is inside
// the window's bounds.
newPointInBounds = MacPtInRect(aQDPoint, &portRect);
```

```
 // Calculate the previous mouse location using the delta info
 aQDPoint.h = (locHIPoint.x - deltaHIPoint.x);
 aQDPoint.v = (locHIPoint.y - deltaHIPoint.y);

 // Check to see if the previous mouse loc for this event is
 // inside the window's bounds.
 oldPointInBounds = MacPtInRect(aQDPoint, &portRect);

 // If both mouse locations are inside the window, or both mouse
 // locations are outside the window then the tracking result is
 // neither Enter nor Exit. The initial tracking guess was
// correct.
 if ( oldPointInBounds != newPointInBounds)
 {
 // One mouse location is in the window and the other
 // is outside the window. If the current mouse loc is in
 // the window then this must be an Enter event, else it
 // must be an Exit event.
 if (newPointInBounds)
 {
 *outResult = kWacomTrackingMouseEntered;
 }
 else
 {
 *outResult = kWacomTrackingMouseExited;
 }
 }

 }
 }

 // Don't forget to set the current port back to what it was before
 // we changed it.
 SetPort(oldPort);
 }

 // If there was an event, invoke the application's event handlers for it.
 SendEventToEventTarget ( eventRef, GetEventDispatcherTarget() );

 // Since we pulled the event off the queue, we have to release it.
 ReleaseEvent( eventRef );
 }

 return result;
 }
```

Cocoa Notes

The AppKit framework's mouse event already has a scaled pressure field and sub-pixel information. The pressure field is a float value scaled from 0.0 to 1.0. The sub-pixel information is stored as the fractional parts of the float x and y mouse locations.

In 10.4, Apple has added native methods to NSEvent that allows you to get to the embedded, or pure, tablet data. If your

Cocoa application is 10.4 only, please review NSEvent.h for the method names.

However, if your apps run on pre 10.3.x or less, and you want any of the other tablet data besides pressure and sub-pixel data, then you must do some trickery. First you need to extend the NSEvent class by creating a Category that allows you to get to the underlying Carbon EventRef. Then you can you can use the above information to extract the embedded tablet information.

Sample Code: This is how to create a Category to extend NSEvent to get at the Carbon Event Ref

```
// Tablet Events.h
#import <Cocoa/Cocoa.h>
@interface NSEvent ( TabletEvents )
- (void *)eventRef;
@end

// Tablet Events.m
#import "TabletEvents.h"
@implementation NSEvent (TabletEvents)
- (void *)eventRef
{
return _eventRef;
}
@end
```

By default, Cocoa will either ignore pure tablet proximity events (10.3.x or less) or send the proximity event to the first responder it can (10.4+). Neither of these cases are the best solution as the object that needs to know about proximity may not get the event. What you should do is subclass NSApplication and override the sendEvent method. If you get any tablet class, tablet proximity events, then extract it and have your NSApplication subclass broadcast the proximity event as a NSNotification and then all interested objects will receive it. If not, then send the event along on it's way via `[super sendEvent:theEvent];`. Do not forget to change the Principal Class in the Target Settings, Application Settings tab, Cocoa Specific sub section, to your NSApplication subclass. This forces Cocoa to create an instance of your subclass instead of NSApplication when the program is launched.

Cocoa still does not have a way to natively listen for background events (as of 10.4.x). This means that your Cocoa application will miss important proximity events while your app is in the background. For example, if the user is erasing in your application and clicks in the Finder, flips the pen over, does some stuff and clicks back into your Cocoa application, your cocoa application will still erase because it did not get the proximity event that told the system the pen was flipped back over. You can get around this in 10.3.x by installing your own Carbon event handlers on the monitor target in your NSApplication subclass. This can get a little nuts if you are only running on 10.4 and using the native Cocoa tablet events because you have to deal with 2 proximity event styles. This isn't terribly hard, but it is a pain. If it really bothers you, let Apple know by filing a bug report at http://bugreporter.apple.com.

Tip: (for 10.3.x or less)
The Cocoa Complex sample project has code that does most of the Carbon stuff for you.

TabletEvents.h - Extends the NSEvent class by creating a Category that allows you to get to the extended Tablet Event data such as Tilt and DeviceID from an NSEvent object.

TabletApplication.h - A subclass of NSApplication that will properly broadcast a proximity event as a Notification.

Tip: (for 10.4.x or higher)
The Tiger Cocoa Complex sample project has code that does most of the Carbon stuff for you.

TabletApplication.h - A subclass of NSApplication that will properly broadcast a proximity event as a Notification.

Advanced Techniques (a)

Sub Pixels

Wacom tablets are very precise, high-resolution devices. It is possible to determine where the cursor should be on the screen at fractional pixel levels. That is, you can tell how far between two pixels the cursor is supposed to be. The distance away from the reported pixel location of the cursor and the actual location of the cursor is called the sub pixel information.

Sub pixel information is particularly helpful if the document is of a higher resolution than the screen. You can use the sub-pixel information to get a cursor location that is much closer to your documents resolution than the screen can possibly be.

You can get the size of the tablet and do the scaling yourself, but Apple has modified the mouse event to make it easy for you. When you extract the mouse location from a mouse carbon event, you would generally ask for a `typeQDPoint`. Instead, you should ask for a `typeHIPoint`. HIPoints use float values instead of integer values and the sub pixel information is stored in the fractional part of float number. A HIPoint is defined in CarbonEvents.h as follows;

```
/*
* HIPoint
*
* Discussion:
* HIPoint is a new, floating point-based type to help express
* coordinates in a much richer fashion than the classic QuickDraw
* points. It will, in time, be more heavily used throughout the
* Toolbox. For now, it is replacing our use of typeQDPoint in mouse
* events. This is to better support sub-pixel tablet coordinates.
* If you ask for a mouse location with typeQDPoint, and the point
* is actually stored as typeHIPoint, it will automatically be
* coerced to typeQDPoint for you, so this change should be largely
* transparent to applications. HIPoints are in screen space, i.e.
* the top left of the screen is 0, 0.
*/
struct HIPoint {

/*
* The horizontal coordinate of the point.
*/
float x;

/*
* The vertical coordinate of the point.
*/
float y;
};
```

Sample Code:

```
EventRef inEvent; // Reference to an existing mouse event
HIPoint aPoint; // This is where the mouse location with sub pixel
// information will be stored.

GetEventParameter(inEvent,
kEventParamMouseLocation,
typeHIPoint,
NULL, sizeof(HIPoint), NULL,
(void *)&aPoint);
```

Cocoa Notes:

The AppKit framework’s mouse event also has sub-pixel information. The sub-pixel information is stored as the fractional parts of the float x and y mouse locations.

Remembering Tool Settings

A really simple but useful feature you can add to your application is the ability to remember settings for each of a user’s tools. For instance, a user may have three Intuos styli and would like to set one to the paint brush, another to pencil, and the third to the clone tool. Then as the user puts one pen down and starts to use one of the other pens, your application could recognize this and automatically switch to the correct tool pre-defined for that pen.

The secret to this feature is the combined use of the uniqeID of the proximity event and the deviceID of the proximity and pointer events. Keep a list all known devices and the settings for each device. Then as you get proximity events, check to see if you already have a device with that uniqueID in the list. If so, then switch to the settings for that device and update the deviceID. If not, then add that device to the list. While you receive pointer events with the same deviceID, you know the user is still using the same tool. On the rare occasion that you start getting pointer events with a different deviceID without first getting a proximity event, simply ask the tablet driver to resend the last proximity event (a.cm). You should also save this list in your preference file so that the next time the user runs your application, the settings for each tool will be remembered.

Important:
You may be tempted to bypass the uniqueID and just use the deviceID. However, Wacom will not guarantee that the deviceID for a tool will remain the same. The uniqueID is guaranteed to remain the same for a transducer at all times. Even if a transducer with a uniqueID of 425 on one computer is used on another computer, it will report its uniqueID as 425 on the other computer.

Performance Issues

Tablets generate more mouse events than typical application can handle. MacOS X helps these applications out by “coalescing” mouse events. That is, if an application has outstanding mouse events when a new one mouse event is generated, the OS will combine the new mouse event with the existing mouse event. This keeps normal applications as responsive as possible. If MacOS X did not do this, just dragging a window with a tablet would cause the window dragging to noticeably lag behind the cursor on the screen. As a tablet aware application, this means that you will probably lose tablet data. In other words, if you are a drawing application, the curves the user draws may not look like curves, but a few very choppy straight lines. You can tell MacOS X to stop coalescing mouse events for just your application using the SetMouseCoalescingEnabled() API call. However, if you do so, you must be prepared for the onslaught of events that your application will receive.

Warning:
With mouse coalescing turned off, your application must be able to handle 100 to 200+ mouse events with embedded tablet data per second! If you your application fails to keep up, your event que can overflow and you may miss important events such as mouse up!

Technique 1: Only turn off coalescing when needed! – Typically, you only need all the tablet events when the user is dragging in a particular area of your application. You already probably have a tracking loop setup for this. When you enter this tracking loop because of a mouse down, turn off mouse coalescing, and then turn it back on again when you exit the tracking loop. Alternatively, you can setup a tracking rect and turn mouse coalescing off when the cursor enters the tracking rect and then turn it back on when the cursor leaves the tracking rect. This way, you can let Carbon or Cocoa handle things like window dragging or button handling without affecting their performance.

Technique 2: Process all outstanding drag events at the same time. When your application has turned mouse coalescing off, processes all outstanding mouse drag events at one time. Simply setup a ReceiveNextEvent() loop with a duration of “no wait” pulling off every mouse drag event from the event que. Then process all of these drag events at the same time then update the screen.

Sample Code:
```
//////////////////////////////////////////////////////////////////////////////
// HandleMyControl
//
// This function is the call back that gets called by the OS to handle events
// sent to the "My Control" control.
//
//////////////////////////////////////////////////////////////////////////////
pascal OSStatus HandleMyControl (EventHandlerCallRef inCallRef,
                                 EventRef inEvent, void *userData )
{
      ControlRef    theControl = NULL;
      WindowRef     window = (WindowRef)userData;

      GetEventParameter(inEvent, kEventParamDirectObject, typeControlRef,
                    NULL, sizeof(ControlRef), NULL, (void *)&theControl);
```

```
	switch(GetEventKind(inEvent))
	{
		case kEventControlClick:
		{
			Boolean oldMouseCoalescing = true;

			// Track the cursor as the user drags it in this control!

			// First, turn off mouse coalescing so that we get ALL the tablet
events
			SetMouseCoalescingEnabled(false,&oldMouseCoalescing);

			MyTabletTracking(theControl, window);

			// The tracking loop has ended, the user must no longer be dragging,
			// set mouse coalescing back to what it was.
			SetMouseCoalescingEnabled(oldMouseCoalescing, NULL);

			return noErr;
		}
		break;

		//case kEventControlDraw: and other control events that you should pay
attention to.
		//break;

		default:
		break;
	}

	return eventNotHandledErr;
}

//////////////////////////////////////////////////////////////////////////////
// MyTabletTracking
//
// This function gets called to track the mouse.
//
//////////////////////////////////////////////////////////////////////////////
void MyTabletTracking(HIPoint startPoint, ...)
{
	const EventTypeSpec	mouseEvents[ 4 ] = { { kEventClassMouse, kEventMouseDown },
						     { kEventClassMouse, kEventMouseUp },
						     { kEventClassMouse, kEventMouseDragged },
						     { kEventClassTablet, kEventTabletPointer
}};
	const EventTypeSpec	dragEvents[ 2 ] = { { kEventClassMouse, kEventMouseDragged },
						     { kEventClassTablet, kEventTabletPointer
}};
	EventRef		currentEvent = NULL;
	EventRef		dragEventQue[kEventQueSize] = {NULL};
	int			dragEventQueIdx = 0;
	Boolean				continueTracking = TRUE;



	// Track the movement of the pen until a mouse up.

	while(continueTracking)
	{
		if(noErr == ReceiveNextEvent( 4, mouseEvents,
					kEventDurationForever, true, &eventRef ))
		{
			// We got a mouse event! What kind?
			if(GetEventKind(currentEvent) == kEventMouseUp)
			{
				// you should probably check to make sure it's the left mouse
that
                              // went up!
                              continueTracking = false;

                              // **** Very Important!!!! *****
                              // Don't forget to release the event.
                              ReleaseEvent(currentEvent);
                      }
                      else
                      {
                              // Store off the first event into the event que
                              dragEventQueIdx = 0;
                              dragEventQue[dragEventQueIdx] = currentEvent;

                              // Get all the drag events that are in the App event que
                              do
                             {
                                     ReceiveNextEvent(2, dragEvents,
                                                    kEventDurationNoWait, true
,&currentEvent);
                                     if(currentEvent)
                                     {
                                            dragEventQueIdx++;
                                            if(dragEventQueIdx < kEventQueSize)
                                            {
                                                   dragEventQue[dragEventQueIdx] =
currentEvent;
                                            }
                                            else
                                            {
                                                   // more drag events than we can handle at
one
                                                   // time. You should probably increase your
                                                   // buffer size, but for now don't let the
app
                                                   // event que overflow

                                                   FlushEventsMatchingListFromQueue(
                                                          GetCurrentEventQueue(), 2,
dragEvents);
                                            }
                                     }
                              }while(currentEvent);

                              // No more drag events in the app que. We got them all.

                              // ProcessListOfDragEventsOfSize(dragEventQue,
dragEventQueIdx);
                              // Do the SLOW process of refreshing the screen

                              // **** Very Important!!!! *****
                              // Don't forget to release the events!
                              dragEventQueIdx = 0;
                              while( dragEventQue[dragEventQueIdx] &&
                                     dragEventQueIdx < kEventQueSize)
                              {
                                     ReleaseEvent(dragEventQue[dragEventQueIdx]);
                                     dragEventQue[dragEventQueIdx] = NULL;
                                     dragEventQueIdx++;
                              }
                      }
               }
       }
}
```

Technique 3: Update the dirty parts of the screen on an interval. The user typically does not need to see a screen refresh for every tablet event. Instead, keep track of the dirty rect and only repaint the dirty portion of the screen 15 to 30 times a second. To the user, it looks like the drawing is keeping pace with the cursor and your app has more time to do process each mouse event.

Other Technique Ideas: Use a separate drawing thread. Draw a “draft” version first and then draw a detailed final version while the user is not drawing. Or combine any of these techniques or use your own technique.

Communicating With the Tablet Driver

As of version 4.7.5 of the Wacom Tablet Driver, all communication with the tablet driver should be done with Apple Events. The target of these Apple Events should be the TabletDriver application.

Important:
Communicating with the Tablet Driver via Apple Events can only be done on MacOS X 10.2 or higher and Wacom’s MacOS X tablet Driver version 4.7.5 or higher.

Tablet Driver Object Tree
cWTDContext
cWTDDriver
|.........cWTDTablet
|.........cWTDCustomizedApp
|.........cWTDPopItem
|.........cWTDMenuItem
|.........cWTDTransducer
|.........cWTDButton

Tip:
All of the tablet Apple Event definitions can be found in the TabletAEDictionary.h file that is included in the Simple Carbon Tablet Events (and Cocoa Complex) sample code. Also the TAEHelpers functions located in the Simple Carbon Tablet Events sample code greatly simplifies the building and sending of Tablet Apple Events to the tablet driver.

Generic *Communication* with the Tablet Driver

The Generic set of Apple Events allows you to programmatically look at settings in the Tablet Driver. Though you can modify settings with these events, these changes would affect the global operation of the Tablet. In general you should not change these settings. The most common use of these events would be to get the version number of the driver, the number of tablets, and the size of each tablet.

Application Specific Communication With the Tablet Driver

The Application Specific Apple Events are composed of a new Wacom Apple Event SendTabletEvent and a new Context class.

You can use the Send Tablet Event (`eSendTabletEvent`) to instruct the driver to re-post the current pointer or proximity event to the system. This is particularly useful in getting the Tablet Driver to send the current proximity event to different controls as their order in the “Event Chain” change. Also if you notice the deviceID of the events, change and you did not process a proximity event for the new deviceID, then you can ask for the proximity event to be resent.

Sample Code:

```
#define cTabletEvent        'TblE'

#define kAEWacomSuite       'Wacm'
#define eSendTabletEvent    'WSnd'
#define eEventProximity     'WePx'
#define eEventPointer       'WePt'
#define kDefaultTimeOut 15 //Timeout value in ticks Approx 1/4 second
```

```
//////////////////////////////////////////////////////////////////////////////
// ResendLastTabletEventofType
// parameters:
// DescType tabletEventType - eEventProximity, eEventPointer
// returns: noErr on success, else an AE error code
//////////////////////////////////////////////////////////////////////////////
OSErr ResendLastTabletEventofType(DescType tabletEventType)
{
OSType        tdSig = kWacomDriverSig;
AEDesc        driverTarget;
AppleEvent       aeSend;
OSErr         err;

   // Create the Target this Apple Event is to be sent to (The Tablet Driver)
   AEInitializeDesc(&driverTarget);
   err = AECreateDesc(typeApplSignature,
                      (Ptr) &tdSig,
                      sizeof(tdSig),
                      &driverTarget);
if(err)
{
AEDisposeDesc(&driverTarget);
return err;
}

err = AECreateAppleEvent(kAEWacomSuite,       // Create a special Wacom Event
eSendTabletEvent, // Send Last Tablet Event
&driverTarget,
kAutoGenerateReturnID,
kAnyTransactionID,
&aeSend);
if(err)
{
AEDisposeDesc(&driverTarget);
return err;
}

  err = AEPutParamPtr ( &aeSend, keyAEData,
                      typeEnumeration,
                      &tabletEventType,
                          sizeof(tabletEventType)); // Add what type of event to send.

// Finally send the event
err = AESend(&aeSend,    // The complete AE we created above
NULL,
kAEWaitReply,
kAEHighPriority,
kDefaultTimeOut,
NULL,
NULL);
AEDisposeDesc(&aeSend);

return err;
 }
```

Context events are how some of the more advanced techniques are accomplished such as automatic scaling, disconnecting the tablet events from the mouse and more. Your application can ask the Tablet Driver to create one or more contexts per tablet and then modify the properties of these contexts as you see fit. These contexts are specific to the instance of your application that created them. Keep in mind that your contexts are only valid while your application is in the foreground. Your contexts will cease to function while you application is in the background. When your application returns as the foreground application, your contexts return as well, When your application quits, the contexts go away permanently. (Though please be nice and destroy any created contexts before quitting.) When your application is re-launched, it must create it's contexts all over again.

To **create** a context, send to the Tablet Driver an Apple Event of class / type `{kAECoreSuite, kAECreateElement}` with the `keyAEObjectClass` Param of the Apple Event filled with a `DescType` of `cContext` and the `keyAEInsertHere` Param filled with an object specifier of the index of the tablet (`cTablet`) you want to create a context for.

To **set** and **get** properties of a context, send to the Tablet Driver an Apple Event of class / Type `{kAECore, kAESetData or kAEGetData}` with the `keyDirect` Apple Event Parameter filled with an object specifier of the context's (`cContext`) uniqueID (`formUniqueID`) and the property's (`cProperty`) ID (`formPropertyID of type DescType`).

Context Property	Value	Type	Description
`pContextMapScreenArea`	`'Smap'`	`typeQDRectangle`	This is the area of the Desktop, in pixels, that this context's tablet area will map to.
`pContextMapTabletOutputArea`	`'Tomp'`	`typeLongRectangle`	This is the rectangle, that this context will map the Tablet Input Area to. ie, the location of the transducer on the tablet input area will be scaled to this range and the result is returned in the absX, and absY tablet pointer event fields.
`pContextMapTabletInputArea`	`'Tmap'`	`typeLongRectangle`	This is the area of the Tablet, in tablet counts, that this context is valid for. When a transducer is in this rectangle on the tablet, it will obey the rules of this context.
`pContextMovesSystemCursor`	`'Mvsc'`	`typeBoolean`	Should events generated from this context move the system cursor? If false, then all tablet events will be sent as pure tablet events.
`pContextEnabled`	`'Cenb'`	`typeBoolean`	Is this context enabled? You can enable and disable the contexts you create At any time.

To **destroy** a context, send to the Tablet Driver an Apple Event of class / Type `{kAECore, kAEDelete}` with the `keyDirect` Apple Event Parameter filled with an object specifier of the context's (`cContext`) uniqueID (`formUniqueID`).

Tip:
The TAEHelpers functions located in the Tablet Event Demo sample code contains functions to Create, Destroy, Read and Modify Contexts and their attributes. Their use is even demonstrated with the "Constrain to Window" menu command.

Scaling

To get the Tablet Driver to scale it's output to a specific section on the screen, create a context and modify the `pContextMapScreenArea` attribute to a rectangle on the screen.

To get the Tablet Driver to scale it's absX, and absY data, create a context and modify the `pContextMapTabletOutputArea` to the output rectangle you would like.

Note that changing the output rectangle does not modify the screen area that the tablet driver moves the cursor and vice versa. So, for example, you can modify the `pContextMapScreenArea` of your context to force the cursor to remain inside your window. Then you can modify the `pContextMapTabletOutputArea` of the same context so that the abs range is 0 – 1000 in both axis. Why would you want to do this? I don't know, but you can!

Dual Track

Dual Track is only supported on Intous1 and Intous2 tablets. This feature was dropped for the Intous3 tablet line.

Overriding Tablet Controls

WacomTabletDriver version 6.1.0 provides a set of Apple Events that enable applications to take control of tablet controls. There are three types of tablet controls: ExpressKeys, TouchStrip, and TouchRing. Each control has one or more functions associated with it. Do not make assumption of the number of controls of a specific tablet or the number of functions associated with a control. Always use the APIs to query for the information.

An application needs to do the following to override tablet controls:

1. Create a context for the tablet of interest.
2. Register with the distributed notification center to receive the overridden controls' data from user actions.
3. Query for number of controls by control type (ExpressKeys, TouchStrip, or TouchRing).
4. Query for number of functions of each control.
5. Enumerate the functions to find out which are available for override.
6. Set override flag for a control function that's available.
7. Handle the control data notifications to implement functionality that the application desires for the control function.
8. **Must** destroy the context upon the application's termination or when the application is done with it.

To create an override context for a tablet, send `to the Tablet Driver an Apple Event of class / type {kAECoreSuite, kAECreateElement} with the keyAEObjectClass` Param of the Apple Event filled with a `DescType` of `cContext`, the `keyAEInsertHere` Param filled with an object specifier of the index of the tablet (`cWTDTablet`) and the keyASPrepositionFor Param filled with a DescType of pContextTypeBlank.

`To destroy a context, send to the Tablet Driver an Apple Event of class / Type {kAECore, kAEDelete} with the keyDirect Apple Event Parameter filled with an object specifier of the context's (cContext) uniqueID (formUniqueID).`

Cocoa Sample Code:

```
#define kWacomDriverSig              'WaCM'
#define cContext                     'CTxt'
#define pContextTypeBlank            'Blnk'


//////////////////////////////////////////////////////////////////////////////
// Helper function that returns Apple Event descriptor of the tablet driver.
//
+ (NSAppleEventDescriptor *)driverAppleEventTarget
{
   OSType tdSig = kWacomDriverSig;
   return [NSAppleEventDescriptor descriptorWithDescriptorType:typeApplSignature
            bytes:&tdSig length:sizeof(tdSig)];
}
//////////////////////////////////////////////////////////////////////////////
// Create a blank context for the tablet with index tabletIndex
//
+ (UInt32)makeContextForTablet:(UInt32)tabletIndex
{
   UInt32 context = 0;
   NSAppleEventDescriptor *response = nil;
   NSAppleEventDescriptor *event = [NSAppleEventDescriptor
            appleEventWithEventClass:kAECoreSuite
            eventID:kAECreateElement
            targetDescriptor:[self driverAppleEventTarget]
            returnID:kAutoGenerateReturnID
            transactionID:kAnyTransactionID];

   [event setDescriptor:[NSAppleEventDescriptor descriptorWithTypeCode:cContext]
            forKeyword:keyAEObjectClass];

   [event setDescriptor:[NSAppleEventDescriptor descriptorForObjectOfType:cWTDTablet
            withKey:[NSAppleEventDescriptor descriptorWithUInt32:tabletIndex]
            ofForm:formAbsolutePosition]
            forKeyword:keyAEInsertHere];

   [event setDescriptor:[NSAppleEventDescriptor descriptorWithTypeCode:pContextTypeBlank]
            forKeyword:keyASPrepositionFor];

   response = [event sendExpectingReplyWithPriority:kAEHighPriority
            andTimeout:kWtcTabletDriverAETimeout];

   context = [[response descriptorForKeyword:keyDirectObject] int32Value];

   return context;
}
```

```
//////////////////////////////////////////////////////////////////////////////
// Destroy context
//
+ (void)destroyContext:(UInt32)context
{
   NSAppleEventDescriptor *event = [NSAppleEventDescriptor
            appleEventWithEventClass:kAECoreSuite
            eventID:kAEDelete
            targetDescriptor:[self driverAppleEventTarget]
            returnID:kAutoGenerateReturnID
            transactionID:kAnyTransactionID];

   [event setDescriptor:[NSAppleEventDescriptor descriptorForObjectOfType:cContext
            withKey:[NSAppleEventDescriptor descriptorWithUInt32:context]
            ofForm:formUniqueID]
            forKeyword:keyDirectObject];

   [event sendWithPriority:kAEHighPriority andTimeout:kWtcTabletDriverAETimeout];
}
```

To Get or Set properties of a control function, send to the Tablet Driver an Apple Event of class / Type {kAECore, kAESetData or kAEGetData} with the keyDirect Apple Event Parameter that specifies the override context, index of the control and function to override. The structure of the descriptor filled looks like this:
cContext
|---àcWTDExpressKey (cWTDTouchRing, or cWTDTouchStrip)
|---à cWTDControlFunction

The table below lists the properties that an application can Get and/or Set for a tablet control or its functions.

Property	Value	Type	Description	Notes
pFunctionAvailable	'FunA'	typeBoolean	This indicates whether the control function is available for an application to override.	Read-only.
pControlMinValue	'CMin'	typeUInt32	The minimum value of the control.	Read-only. Function index ignored.
pControlMaxValue	'CMax'	typeUInt32	The maximum value of the control.	Read-only. Function index ignored.
pControlLocation	'CLoc'	typeUInt32	The physical location of the control on the tablet.	Read-only. Function index ignored. 0 - left; 1 - right; 2 – top; 3 – bottom.
pOverrideFlag	'OvrF'	typeBoolean	Get or set the override flag for a control function.	Read/Write.

Cocoa Sample Code:

```
#define kInavlidAppleEventIndex     0

#define cContext                   'CTxt'
#define cWTDTouchRing              'WRnG'
#define cWTDExpressKey                    'WExK'
#define cWTDTouchStrip                    'WTcS'
#define cWTDControlFunction        'WCtF'
```

```
#define pFunctionAvailable          'FunA'
#define pOverrideFlag               'OvrF'
#define pOverrideName               'ONme'

typedef enum eAETabletControlType
{
   eAETouchRing = 0,
   eAETouchStrip,
   eAEExpressKey
} eAETabletControlType;

//////////////////////////////////////////////////////////////////////////////
// Helper function for translating tablet control type to Apple Event class.
//
+ (DescType)descTypeFromControlType:(SInt32)controlType
{
   if (controlType == eAETouchStrip)
   {
    return cWTDTouchStrip;
   }
   else if (controlType == eAEExpressKey)
   {
    return cWTDExpressKey;
   }
   return cWTDTouchRing;
}

//////////////////////////////////////////////////////////////////////////////
// Helper function for querying the tablet driver for a tablet control property.
//
+ (NSAppleEventDescriptor*)dataForAttribute:(DescType)attribute ofType:(DescType)dataType
         ofFunction:(UInt32)function ofControl:(UInt32)control
         ofContext:(UInt32)context forControlType:(SInt32)controlType
{
   NSAppleEventDescriptor *reply = nil;
   NSAppleEventDescriptor *event = [NSAppleEventDescriptor
         appleEventWithEventClass:kAECoreSuite
         eventID:kAEGetData
         targetDescriptor:[self driverAppleEventTarget]
         returnID:kAutoGenerateReturnID
         transactionID:kAnyTransactionID];

   // context descriptor
   NSAppleEventDescriptor *contextDesc = [NSAppleEventDescriptor
         descriptorForObjectOfType:cContext
         withKey:[NSAppleEventDescriptor descriptorWithUInt32:context]
         ofForm:formUniqueID];

   // control descriptor
   NSAppleEventDescriptor *controlDesc = [NSAppleEventDescriptor
         descriptorForObjectOfType:[self descTypeFromControlType:controlType]
         withKey:[NSAppleEventDescriptor descriptorWithUInt32:control]
         ofForm:formAbsolutePosition from:contextDesc];

   // function descriptor
   NSAppleEventDescriptor *functionDesc = (kInavlidAppleEventIndex != function) ?
     [NSAppleEventDescriptor
            descriptorForObjectOfType:cWTDControlFunction
            withKey:[NSAppleEventDescriptor descriptorWithUInt32:function]
            ofForm:formAbsolutePosition from:controlDesc] : nil;

   // attribute descriptor
   NSAppleEventDescriptor *attribDesc = [NSAppleEventDescriptor
         descriptorForObjectOfType:formPropertyID
         withKey:[NSAppleEventDescriptor descriptorWithTypeCode:attribute]
         ofForm:formPropertyID from:functionDesc ? functionDesc : controlDesc];

   [event setDescriptor:attribDesc forKeyword:keyDirectObject];
   [event setDescriptor:[NSAppleEventDescriptor descriptorWithTypeCode:dataType]
         forKeyword:keyAERequestedType];

   // send
   reply = [event sendExpectingReplyWithPriority:kAEHighPriority
         andTimeout:kWtcTabletDriverAETimeout];

   return [reply descriptorForKeyword:keyDirectObject];
}

////////////////////////////////////////////////////////////////////////////////
// Helper function for setting a tablet control property.
//
+ (BOOL)setBytes:(void*)bytes ofSize:(UInt32)size ofType:(DescType)dataType
         forAttribute:(DescType)attribute
         ofFunction:(UInt32)function ofControl:(UInt32)control
         ofContext:(UInt32)context forControlType:(SInt32)controlType
{
   OSErr err = noErr;
   NSAppleEventDescriptor *data = nil;
   NSAppleEventDescriptor *event = [NSAppleEventDescriptor
         appleEventWithEventClass:kAECoreSuite
         eventID:kAESetData
         targetDescriptor:[self driverAppleEventTarget]
         returnID:kAutoGenerateReturnID
         transactionID:kAnyTransactionID];

   // context descriptor
   NSAppleEventDescriptor *contextDesc = [NSAppleEventDescriptor
         descriptorForObjectOfType:cContext
         withKey:[NSAppleEventDescriptor descriptorWithUInt32:context]
         ofForm:formUniqueID];

   // control descriptor
   NSAppleEventDescriptor *controlDesc = [NSAppleEventDescriptor
         descriptorForObjectOfType:[self descTypeFromControlType:controlType]
         withKey:[NSAppleEventDescriptor descriptorWithUInt32:control]
         ofForm:formAbsolutePosition from:contextDesc];

   // function descriptor
   NSAppleEventDescriptor *functionDesc = (kInavlidAppleEventIndex != function) ?
     [NSAppleEventDescriptor
            descriptorForObjectOfType:cWTDControlFunction
            withKey:[NSAppleEventDescriptor descriptorWithUInt32:function]
            ofForm:formAbsolutePosition from:controlDesc] : nil;

   // attribute descriptor
   NSAppleEventDescriptor *attribDesc = [NSAppleEventDescriptor
         descriptorForObjectOfType:formPropertyID
         withKey:[NSAppleEventDescriptor descriptorWithTypeCode:attribute]
         ofForm:formPropertyID
         from:functionDesc ? functionDesc : controlDesc];

   [event setDescriptor:attribDesc forKeyword:keyDirectObject];

   [event setDescriptor:[NSAppleEventDescriptor descriptorWithTypeCode:dataType]
         forKeyword:keyAERequestedType];

   data = [NSAppleEventDescriptor descriptorWithDescriptorType:dataType
         bytes:bytes length:size];

   [event setDescriptor:data forKeyword:keyAEData];

   // send
   err = [event sendWithPriority:kAEHighPriority andTimeout:kWtcTabletDriverAETimeout];

   return (err == noErr);
}
```

```
/////////////////////////////////////////////////////////////////////////////
// Override touch ring of a tablet associated with the input context.
//
- (void)takeControlOfRingInContext:(UInt32)context
      withRingCount:(UInt32)ringCount andFunctionCount:(UInt32)functionCount
{
   UInt32 controlIndex;
   for (controlIndex = 1; controlIndex <= ringCount; controlIndex++)
   {
    UInt32 functionIndex;
    for (functionIndex = 1; functionIndex <= functionCount; functionIndex++)
    {
      // check if this function is available for override
      NSAppleEventDescriptor *aeResponse = [[self class]
         dataForAttribute:pFunctionAvailable
         ofType:typeBoolean
         ofFunction:functionIndex
         ofControl:controlIndex
         ofContext:context
         forControlType:eAETouchRing];
      if ([aeResponse booleanValue] != 0)
      {
         // this ring function is available for override
         // send apple event to override
         Boolean overrideFlag = 1;
         BOOL success = [[self class] setBytes:&overrideFlag
            ofSize:sizeof(overrideFlag) ofType:typeBoolean
            forAttribute:pOverrideFlag
            ofFunction:functionIndex
            ofControl:controlIndex
            ofContext:context
            forControlType:eAETouchRing];
         if (!success)
         {
            NSLog(@"Failed to override Ring#%d Function#%d.",
                      controlIndex, functionIndex);
         }
      }
    }
   }
}
```

An application that overrides tablet controls needs to register with the distributed notification center to receive overridden controls' data when user performs an action on the controls. The user info dictionary of the notifications contains the value of the control and information that indicates which tablet, control and function the data is for:

1. Tablet index – 1-based index for the tablet of interest.
2. Control type - the type of the tablet control.
3. Control index – 1-based index for the tablet control.
4. Function index – 1-based index for the currently active function of the control.
5. Control value – current value of the control.

Cocoa Sample Code:

```
#define kWacomNotificationObject           "com.wacom.tabletdriver.hardware"
#define kWacomTabletControlNotification    "com.wacom.tabletdriver.hardware.controldata"

#define kTabletNumberKey     "Tablet Number"      // 1-based system tablet index
#define kControlTypeKey      "Control Type"       // eAETouchRing, eAETouchStrip, or
eAEExpressKey
#define kControlNumberKey    "Control Number"     // 1-based index
#define kFunctionNumberKey   "Function Number"    // 1-based index
#define kControlValueKey     "Control Value"      // current control value
```

```
//////////////////////////////////////////////////////////////////////////////
// Register observer for control data notificaations.
//
- (void)observeContext:(UInt32)context
{
   // NOTE: suspend the notifications when the application is in the background
   // since the overrides are effective only when the application is in the foreground
   if (context != 0)
   {
    [[NSDistributedNotificationCenter
notificationCenterForType:NSLocalNotificationCenterType]
            addObserver:self
            selector:@selector(tabletControlData:)
            name:@kWacomTabletControlNotification
            object:@kWacomNotificationObject
            suspensionBehavior:NSNotificationSuspensionBehaviorDrop];
   }
}

//////////////////////////////////////////////////////////////////////////////
// This is where we handle tablet control data notification.
//
- (void)tabletControlData:(NSNotification*)note
{
   // extract control data from the notification
   NSDictionary *userInfo = [note userInfo];
   SInt32 controlType = [[userInfo objectForKey:@kControlTypeKey] longValue];
   UInt32 tabletIndex = [[dict objectForKey:@kTabletNumberKey] unsignedLongValue];
   UInt32 controlIndex = [[dict objectForKey:@kControlNumberKey] unsignedLongValue];
   UInt32 functionIndex = [[dict objectForKey:@kFunctionNumberKey] unsignedLongValue];
   UInt32 controlValue = [[dict objectForKey:@kControlValueKey] unsignedLongValue];

   // do something with the data
   // ...
}
```