Notes for Mac Tablet Aware Application Developers

Audience

This document is mainly intended to assist developers who already have tablet aware applications and want to add capabilities to their applications. You should already be able to access the tablet data structure and it's elements. Developers who are adding tablet features for the first time should start with the **Overview** section, and examine the sample code.

Overview

There are four points in an application where modifications may be made to support tablets:

1) Initialization

During initialization you need to:

a) Determine if a tablet is attached. If one is attached, get a pointer to the tablet's data structure. All references to tablet data is through this structure.

b) Get static information about the tablet, such as tablet dimensions and coordinate resolution.

2) Event Loop - Enter Proximity (OS) Events

In the applications event loop, look for a tablet enter proximity event. (Tablet proximity events are described in detail in the **Eraser** section below.) Here, you will find information about the tablet device currently in use. This information can be used to change tools or screen cursors, for example, when a stylus eraser is placed in use or when tools are changed based on device ID. Other information, such as the number of pressure levels and whether or not tilt and rotation are supported on the device, can be grabbed for later use when drawing. Some of this data can be directly accessed in the draw loop.

3) Event Loop - Mouse Down (draw loop)

In the draw loop (usually continuously done between mouse down and mouse up), tablet data is used to modify the rendering of the line being drawn. Data such as current pressure, tilt angles, and fingerwheel values are used to change rendering parameters like opacity, brush direction, line thickness, and so forth.

4) Event Loop - Null events

During a null task you may want to look at device information for navigational purposes. For example, if you are using a 4D Mouse for navigation, current device information can be used to modify a rotating object's position or to modify your own viewpoint.

Notes

Two data structures are used. One follows the Apple standard structure in technical note DV-1 (originally #266). The other follows the Wacom structure and differs slightly from the Apple structure. Both structures are maintained for legacy reasons. You can determine which one you are using by the way that you access it. The Apple standard structure is accessed by "walking" the ADB table or by opening the .APD driver. Most CAD applications use the Apple standard structure. The Wacom structure is accessed by opening the .Wacom driver. Most graphics applications use the Wacom structure.

The structures described in this document have a version number of 3. The version number is in the header section of the structure at record->version;

In the sample code below the pointer to the data structure is named record. Unless noted, this could be a pointer to either the Apple or the Wacom structure.

If you are using the Wacom structure, you will have to change your existing code. This is due to the addition of the ability to track a second device. This has been implemented in the Apple standard data structure for several years now, but has only been used by one developer experimentally. The capability to track two devices is now included in the Wacom data structure. So you will have to change

From:

```
myData = record->someTransducerData;
```

To:

```
myData = record->transducer[which].someTransducerData;
```

As you can see, code is shown in `Courier 10` font.

The device number is named `which`. Valid values of `which` are 0 or 1. The first device to enter proximity is device 0, the second is device 1. Unless you are implementing two handed input, you will set which to zero. See **Multi Device Tracking** below for more details. Pointers to data in the header section, such as pointers to number of pressure levels, do not need to change. This will not break current implementations.

It is best to have at least two undo buffers, or one more than the desired number. Don't flush the undo buffer on a button down when you are about to draw, it takes too long. Flush the buffer on a button up. This will improve the beginning of your strokes.

Features You Should Already Have Implemented

Eraser, Tilt, and Multi Device Tracking are features that have been around for some time. A large installed base of tablets supports these capabilities and, if you have not already done so, you can increase the power and performance of your application by implementing these features.

• Eraser

The obvious purpose of the eraser is to use it as an erasing tool. But you can change the function of the eraser to perform more advanced tasks. Using the stylus tip in conjunction with the eraser, you can easily perform two opposing functions such as dodge and burn, or ink and bleach. The eraser can also function independently as any tool selected in the application.

When the eraser was introduced, a new event was created (the idea originally came from some work done at Apple). Some unused bits in the OS Event event message field were used to indicate a tablet proximity event and whether the device was entering or leaving proximity. (Proximity means the device is within the sensing range of the tablet.) The actual event is an OS event which is broken down into sub-events based on the message field. The tablet event is a sub-event of type mouse moved. This message is normally generated by setting a region when posting a waitNextEvent. Doing double duty with this event usually causes no problems.

So, a tablet proximity event is an OS event. The message of the event has the high byte set to 0xFE to indicate a mouse moved event. If bit 3 is set (0x08), it is a tablet event. If bit 3 is clear, it is a true mouse moved event. If it is a tablet event, then bit 2 (0x04) will be set when entering proximity and clear when leaving.

So what does this really mean? It is to keeps your application event driven. Currently, you probably only examine tablet data in your draw loop, entered by a mouse down event. The mouse moved event is to alert you to check the tablet to see which end of the device entered proximity and when it leaves.

```
// Part of the event loop

switch(theEvent->what)

{

// all events of type OSEvent

case osEvt:

// get the event sub-type from the high byte

theOSEvent = ((theEvent->message >> 24) & 0x0FF);

switch(theOSEvent)

{

// OSEvent of type mouse moved

case mouseMovedMessage:

// now see if it was a mouse moved or a new tablet event

if(theEvent->message & TABLET_EVENT)

{

// It is a tablet event so was it enter or leave prox

if (theEvent->message & ENTER_PROX)

{


//multi tracking // entering proximity - see which transducer entered

// handleMultiTracking();

// OK - so a transducer entered prox.

// Now we need to see which one to choose a tool

switch (record->transducer[which].DOFTrans

& TRANS_MASK)

{

case PEN:

// Switch to the stylus tip tool

oldTool = tool;


tool = tipTool;

break;

case ERASER:

// Switch to the stylus eraser tool

oldTool = tool;

tool = eraserTool;

break;

default:

// Switch to the puck tool

oldTool = tool;

tool = puckTool;

break;

} // endswitch transducer type


// ID // ID GOES HERE - OR IN THE CASES ABOVE

serialNumberHigh = record->transducer[which].highSN;

serialNumberLow = record->transducer[which].lowSN;

// switch tools here based on the serial number

}

else

{
```

```
// The transducer left proximity so switch back

tool = oldTool;

} // endif entered proximity

}

else

{

// It was a real mouse moved event

// Do mouse moved stuff here

} // endif tablet event

case other OS events: go here (suspend/resume)

} // endswitch OS event

case other events: go here (mouse up/down, etc.)

} // endswitch events
```

• Tilt

Stylus tilt has several uses, but the most obvious is to change brush shape. A good example of using stylus tilt to change brush shape is the Wacom Pen Tools Virtual Airbrush tool. Tilt can also be used for object rotation, or as a joystick.

Note that the structure header contains the range of tilt values in `angleRes`, but that this is the full range of values, that is, the total number of discrete values. Since the values reported are from -60 to +60 `angleRes` will contain 120 (total number of values) not 60 (highest value reported).

```
// This probably goes in your enter proximity event section.

// Check for the existence of tilt with:

if ((record->transducer[which].DOFTrans & DOFMASK) == TILT)

{

// Check for the range of tilt values with:
```



```
xTilt = record->transducer[which].xTilt;

yTilt = record->transducer[which].yTilt;

// then scale the tilt to get full effect at full tilt
xTiltEffect = (xTilt * 100%) / tiltRange;

yTiltEffect = (yTilt * 100%) / tiltRange;
```

• Multi Device Tracking

Multi Device Tracking, also referred to as multimode, allows two devices to be tracked at the same time. With the UDII series 12 x 12 and larger it is possible to simultaneously track a stylus and a puck. With the Intuos series it is possible to track ANY two devices.

Working with two tools at once expands your capabilities in many unique ways. Stretch a line from both ends, or move a frisket with one hand while airbrushing around it with the other. Move a magnifying glass with one hand and draw on either the magnified, or the unmagnified image with the other. See the paper Bier, Eric A., Stone, Maureen C., Pier, Ken, Buxton, William. "Toolglass and Magic Lenses: The See-Through Interface" Computer Graphics, (1993), 73-80 or the video tape submission *A GUI Paradigm Using Tablets, Two Hands and Transparency* George Fitzmaurice, Thomas Baudel, Gordon Kurtenbach and Bill Buxton, CHI 97 Video Program by ACM.

Here are some things to keep in mind when implementing multi device tracking:

a) It is not possible to have the system move two screen cursors for you, it can only move one. So you will have to turn tracking on for one device and off for the other (or turn both off). You will have to move your own "screen cursor (sprite?)" for at least one device.

b) There is a major hand and a minor hand, the major hand, being your dominant hand, usually performs the more complex task. For example, the major hand does the drawing while the minor hand may hold a frisket.

c) Avoid collisions. The application should change the offsets for the two devices (or partition the tablet) so that you do not have to position the two devices on top of one another.

By default, tracking is turned on (S bit in `flags` = 0) for device 0 (`which` = 0). This is the first device to enter proximity. Tracking is turned off (S bit = 1) for device 1, the second to enter proximity. So, an application that does not support multi device tracking will virtually have the second device turned off. If they are both on, the screen cursor jumps back and forth between them. You may need to reverse this on the fly (on enter proximity) based on which device is in which hand.

In the **Eraser** section above, there is a point in the code snippet marked **//multi tracking** in bold. This is where you need to track which device came in first and which came in second in order to turn tracking on and off for the appropriate hand.

```
// Initialization portion

// I will save these to restore on suspend and quit events

oldXscale0 = record->transducer[0].xScale;

oldXscale1 = record->transducer[1].xScale;

oldYscale0 = record->transducer[0].yScale;

oldYscale1 = record->transducer[1].yScale;

oldXtrans0 = record->transducer[0].xTrans;

oldXtrans1 = record->transducer[1].xTrans;

oldYtrans0 = record->transducer[0].yTrans;

oldYtrans1 = record->transducer[1].yTrans;
```

```
// I'll need these later, so I computed them here
// I offset the two to avoid collisions
// I also use a 1/4 inch inset margin
// The margin around the edge makes navigation
// near the edge easier
leftInset1 = record->spaceRes >> 2;
// Margin to allow for 80% of surface for each device
// This means a 20% offset between the two devices
leftInset2 = (xDimension - record->spaceRes >> 1);
leftInset2 = (leftInset2 << 1) / 10;
// scale for that 80%
scale = (xDimension - record->spaceRes >> 1);
scale =
// now I'm going to put these into effect
// this avoids the scaling changing later
record->transducer[0].xTrans = leftInset1;
record->transducer[1].xTrans = leftInset2;
record->transducer[0].xScale = scale;
record->transducer[1].xScale = scale;
// Enter proximity event portion.
void HandleMultiTracking(void)
{
if ((record->transducer[0].flags & PROXIMITY)
&& (record->transducer[1].flags & PROXIMITY))
{ // both are in prox now
// Somehow decide which device is in the right and left
// hands such as by device type or ID
majorHand = 0;
minorHand = 1;
// offset the two to avoid collisions
record->transducer[0].xTrans = leftInset1;
record->transducer[1].xTrans = leftInset2;
// turn tracking on on major hand, off on minor
record->transducer[majorHand].flags &= !TRACKING;
record->transducer[minorHand].flags |= TRACKING;
} // endif both transducers in prox
} // end HandleMultiTracking()
```

New Features to Implement

New features being introduced with the Intuos series tablets include: Device ID, 4D Mouse Rotation, 4D Mouse Thumbwheel, and Airbrush Fingerwheel. If your application already supports tablets these features are VERY easy to add.

• Device ID

A new feature to the Intuos series is Device ID. A chip with a unique number is inside every device so every device can be uniquely identified. With Device 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.

The ID code from the device is in two sections. It is the combination of the two that is guaranteed unique. One section, the `highSN`, is actually a code unique to each device type. The other section, the lowSN, is a unique 32 bit number within a device type. `lowSN` may be repeated between device types, but not within a type.

The highSN is a coded bit field. Some bits have special meaning to the hardware, but are not of interest to a developer.

What is of interest is:

There are 12 bits total.

The upper 4 bits, and a couple 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 type are:

General stylus: `(highSN & 0x0F06) == 0x0802`

Airbrush: `(highSN & 0x0F06) == 0x0902`

4D Mouse: `(highSN & 0x0F06) == 0x0004`

5 button puck: `(highSN & 0x0F06) == 0x0006`

To create the unique ID just concatenate highSN (without masking with 0x0F06) and lowSN to make a 48 bit (you will probably use 64 bit) serial number.

This code goes into the enter prox. code , probably at **// ID** in the sample above in **Eraser**.

```
// this probably goes in you mouse moved/enter prox routine

if ((record->version) > 2)

{

switch (record->transducer[which].highSN & 0x0F06)

{

case 0x0902: // this is the airbrush

if ((record->transducer[which].DOFTrans & TRANS_MASK) == PEN)

{

// this is the tip of the airbrush

}

else

{

// this is the eraser of the airbrush

} // endif transducer type

// This could go at the end of every case

// for the ID of this airbrush

airbrushSN = lowSN;

// or for a totally unique ID

uniqueSN = (highSN << 32) + lowSN;

break;

case 0x0802: // this is the general stylus

if ((record->transducer[which].DOFTrans & TRANS_MASK) == PEN)

{

// this is the tip of the general stylus

}

else

{

// this is the eraser of the general stylus

} // endif transducer type

break;

case 0x0006: // this is the 5 button puck

break;

case 0x0004: // this is the 4D Mouse

break;

} // endswitch high serial number

}

else // too old to support ID

{

switch (record->transducer[which].DOFTrans & TRANS_MASK)

{

case PEN: // this is an old general stylus

break;

case ERASER: // this is an old general stylus - eraser end

break;

default: // this is an old puck - 4 or 16 button

break;

} // endswitch transducer type

} // endif new version
```

• Airbrush Fingerwheel

The Intuos series tablets have a new device that is similar to an airbrush. It not only supports pressure on the tip, but also has a fingerwheel on the side to simultaneously vary a second value. For example, using the fingerwheel, you can vary line width with pressure and ink density.

The value of the fingerwheel on the side of the Airbrush is reported in the tangPress (tangential pressure) field. The fingerwheel snaps to the middle when released. The middle is 0, and the full range is `-pressLevels` to `+pressLevels`. `tangPress` has a 0 if there is no support for the fingerwheel.

```
// This goes in your draw routine

// Get the tangential pressure with:

if (record->transducer[which].DOFTrans & TAN_PRESSURE)

{

tangential = record->transducer[which].tangPress;

// then scale the pressure to get full effect at full pressure

tangentialEffect = (tangential * 100%) / pressLevels;

}
```

- **4D Mouse Rotation**

The Intuos series tablets have a new device that is similar to a puck. In addition to normal puck parameters, the 4D Mouse supports axial rotation and a thumbwheel on the side. For example, using the 4D Mouse you can rotate the paper, or an object, with the rotation, and zoom with the thumbwheel.

4D Mouse axial rotation is reported in the new rotation field. Values in this field range from 0 to 360 degrees. Since resolution may increase in the future, the value is represented in 10.6 fixed point notation. The upper 10 bits represent unsigned positive integer degrees. The lower 6 bits represent fractional degrees. With this representation you do not need to consider the resolution or range of values.

```
// This goes in your draw routine

if (record->transducer[which].orientFlag & ROTATION)

{

// Get the rotation with:

rotation = record->transducer[which].rotation;

// then scale the rotation to get 360 degrees

float rotationEffect = rotation;

rotationEffect \= 64;

}
```

• 4D Mouse Thumbwheel

A possible use of the 4D Mouse thumbwheel is for zooming or for 3D navigation. The thumbwheel value is reported in the `zCoord` field. The thumbwheel varies from `zCoord = 1024` when pushed forward to `zCoord = -1024` when pulled back and snaps to the middle when released. The middle is 0, and the total range is `zDimension`. `zDisplace` is `1024` to indicate the negative offset. The values `zDimension` and `zDisplace` are 0 if the tablet does not support the 4D Mouse (such as legacy tablets). The `DOFZ` bit in the `DOFTrans` byte is set if the 4D Mouse is in use. The thumbwheel may not have a broad enough range for all uses. It is left as an exercise to the reader to devise a way to extend the range.

```
 // This goes in your draw routine

if (record->transducer[which].DOFTrans & DOFZ)

{

// Get the thumbwheel with:

wheelPos = record->transducer[which].zCoord;

// then scale the thumbwheel to get full effect at end of range

wheelEffect = (wheelPos * 100%) / record->zDisplace;

}
```