Wacom Technology Corp.

Wintab™ Interface Specification 1.4:

16-bit and 32-bit API Reference

Revised October, 2010

This specification was developed in response to a perceived need for a standardized programming interface to digitizing tablets, three dimensional position sensors, and other pointing devices by a group of leading digitizer manufacturers and applications developers. The availability of drivers that support the features of the specification will simplify the process of developing Windows application programs that incorporate absolute coordinate input, and enhance the acceptance of advanced pointing devices among users.

Address questions and comments to:

Wacom Technology Corporation
1311 SE Cardinal Court
Vancouver, WA 98683
USA Tel: ++1-360-896-9833

- For Wintab programming developer support questions, send email to: **developer@wacom.com**
- For Wintab samples and documentation, visit: http://www.wacomeng.com/windows/index.html
- For general questions regarding Wacom tablet usage, visit: http://www.wacom.com/productsupport/

* Trademark holders: Wintab™ - LCS/Telegraphics, Windows™ and Pen Windows™ - Microsoft Corp. Registered trademark holders: Microsoft® - Microsoft Corp.

Contents

1. Background Information

This document describes a programming interface for using digitizing tablets and other advanced pointing devices with Microsoft Windows Version 3.0 and above. The design presented here is based on the input of numerous professionals from the pointing device manufacturing and Windows software development industries.

In this document, the words "tablet" and "digitizer" are used interchangeably to mean all absolute pointing or digitizing devices that can be made to work with this interface. The definition is not limited to devices that use a physical tablet. In fact, this specification can support devices that combine relative and absolute pointing as well as purely relative devices.

The following sections describe features of tablets and of the Windows environment that helped motivate the design.

1.1. Features of Digitizers

Digitizing tablets present several problems to device interface authors.

- Many tablets have a very high report rate.
- Many tablets have many configurable features and types of input information.
- Tablets often control the system cursor, provide additional digitizing input, and provide template or macro functions.

1.2. The Windows Environment

Programming for tablets in the Windows environment presents additional problems.

- Multitasking means multiple applications may have to share the tablet.
- The tablet must also be able to control the system cursor and/or the pen (in Pen Windows).
- The tablet must work with legacy applications, and with applications written to take advantage of tablet services.
- The tablet driver must add minimal speed and memory overhead, so as many applications as possible can run as efficiently as possible.
- The user should be able to control how applications use the tablet. The user interface must be efficient, consistent, and customizable.

2. Design Goals

While the tablet interface design must address the technical problems stated above, it must also be useful to the programmers who will write tablet programs, and ultimately, to the tablet users. Four design goals will help clarify these needs, and provide some criteria for evaluating the interface specification. The goals are user control, ease of programming, tablet sharing, and tablet feature support.

2.1. User Control

The user should be able to use and control the tablet in as natural and easy a manner as possible. The user's preferences should take precedence over application requests, where possible.

Here are questions to ask when thinking about user control as a design goal:

- Can the user understand how applications use the tablet?
- Is the interface for controlling tablet functions natural and unobtrusive?
- Is the user allowed to change things that help to customize the work environment, but prevented from changing things over which applications must have control?

2.2. Ease of Programming

Programming is easiest when the amount of knowledge and effort required matches the task at hand. Writing simple programs should require only a few lines of code and a minimal understanding of the environment. On the other hand, more advanced features and functions should be available to those who need them. The interface should accommodate three kinds of programmers: those who wish to write simple tablet programs, programmers who wish to write complex applications that take full advantage of tablet capabilities, and programmers who wish to provide tablet device control features. In addition, the interface should accommodate programmers in as many different programming languages, situations, and environments as possible.

Questions to ask when thinking about ease of programming include:

- How hard is it to learn the interface and write a simple program that uses tablet input?
- Can programmers of complex applications control the features they need?
- Are more powerful tablet device control features available?
- Can the interface be used in different programming environments?
- Is the interface logical, consistent, and robust?

2.3. Tablet Sharing

In the Windows environment, multiple applications that use the tablet may be running at once. Each application will require different services. Applications must be able to get the services they need without getting in each others' way.

Questions to ask when thinking about tablet sharing include:

- Can tablet applications use the tablet features they need, independent of other applications?
- Does the interface prevent a rogue application from "hijacking" the tablet, or causing deadlocks?
- Does the sharing architecture promote efficiency?

2.4. Tablet Feature Support

The interface gives standard access to as many features as possible, while leaving room for future extensions and vendor-specific customizations. Applications should be able to get the tablet information and services they want, just the way they want them. Users should be able to use the tablet to set up an efficient, comfortable work environment.

Questions to ask when thinking about tablet feature support include:

- Does the interface provide the features applications need? Are any commonly available features not supported?
- Does the interface provide what users need? Is anything missing?
- Are future extensions possible and fairly easy?
- Are vendor-specific extensions possible?

3. DESIGN CONCEPTS

The proposed interface design depends on several fundamental concepts. *Devices* and *cursor types* describe physical hardware configurations. The interface publishes read-only information through a single *information interface*. Applications interact with the interface by setting up *tablet contexts* and consuming *event packets*. Applications may assume interface and hardware control functions by becoming *tablet managers*. The interface provides explicit support for future *extensions*.

3.1. Device Conventions

The interface provides access to one or more *devices* that produce pointing input. Devices supported by this interface have some common characteristics. The device must define an absolute or relative coordinate space in at least two dimensions for which it can return position data. The device must have a pointing apparatus or method (such as a stylus, or a finger touching a touch pad), called the *cursor*, that defines the current position. The cursor must be able to return at least one bit of additional state (via a button, touching a digitizing surface, etc.).

Devices may have multiple *cursor types* that have different physical configurations, or that have different numbers of buttons, or return auxiliary information, such as pressure information. Cursor types may also describe different optional hardware configurations.

The interface defines a standard orientation for reporting device native coordinates. When the user is viewing the device in its normal position, the coordinate origin will be at the lower left of the device. The coordinate system will be right-handed, that is, the positive x axis points from left to right, and the positive y axis points either upward or away from the user. The z axis, if supported, points either toward the user or upward. For devices that lay flat on a table top, the x-y plane will be horizontal and the z axis will point upward. For devices that are oriented vertically (for example, a touch screen on a conventional display), the x-y plane will be vertical, and the z axis will point toward the user.

3.2. Device Information

Any program can get descriptive information about the tablet via the **WTInfo** function. The interface specifies certain information that must be available, but allows new implementations to add new types of information. The basic information includes device identifiers, version numbers, and overall capabilities.

The information items are organized by *category* and *index* numbers. The combination of a category and index specifies a single information data item, which may be a scalar value, string, structure, or array. Applications may retrieve single items or whole categories at once.

Some categories are *multiplexed*. A single category code represents the first of a group of identically indexed categories, one for each of a set of similar objects. Multiplexed categories include those for devices and cursor types. One constructs the category number by adding the defined category code to a zero-based device or cursor identification number.

The information is read-only for normal tablet applications. Some information items may change during the course of a Windows session; tablet applications receive messages notifying them of changes in tablet information.

3.3. Tablet Contexts

Tablet *contexts* play a central role in the interface; they are the objects that applications use to specify their use of the tablet. Contexts include not only the physical area of the tablet that the application will use, but also information about the type, contents, and delivery method for tablet events, as well as other information. Tablet contexts are somewhat analogous to display contexts in the GDI interface model; they contain context information about a specific application's use of the tablet.

An application can open more than one context, but most only need one. Applications can customize their contexts, or they can open a context using a default context specification that is always available. The **WTInfo** function provides access to the default context specification.

Opening a context requires a window handle. The window handle becomes the context's *owner* and will receive any window messages associated with the context.

Contexts are remotely similar to screen windows in that they can physically overlap. The tablet interface uses a combination of context overlap order and context attributes to decide which context will process a given event. The topmost context in the overlap order whose input context encompasses the event, and whose event masks select the event, will process the event. (Note that the notion of overlap order is separate from the notion of the physical z dimension.) *Tablet managers* (described below) provide a way to modify and overlap contexts.

3.4. Event Packets

Tablet contexts generate and report tablet activity via *event packets.* Applications can control how they receive events, which events they receive, and what information they contain.

Applications may receive events either by polling, or via Windows messages.

- Polling: Any application that has opened a context can call the **WTPacketsGet** function to get the next state of the tablet for that context.

- Window Messages: Applications that request messages will receive the WT_PACKET message (described below), which indicates that something happened in the context and provides a reference to more information.

Applications can control which events they receive by using *event masks*. For example, some applications may only need to know when a button is pressed, while others may need to receive an event every time the cursor moves. Tablet context event masks implement this type of control.

Applications can control the contents of the event packets they receive. Some tablets can return data that many applications will not need, like button pressure and three dimensional position and orientation information. The context object provides a way of specifying which data items the application needs. This allows the driver to improve the efficiency of packet delivery to applications that only need a few items per packet.

Packets are stored in context-specific *packet queues* and retrieved by explicit function calls. The interface provides ways to peek at and get packets, to query the size and contents of the queue, and to re-size the queue.Tablet Managers

The interface provides functions for tablet management. An application can become a tablet manager by opening a tablet *manager handle*. This handle allows the manager access to special functions. These management functions allow the application to arrange, overlap, and modify tablet contexts. Managers may also perform other functions, such as changing default values used by applications, changing ergonomic, preference, and configuration settings, controlling tablet behavior with non-tablet aware applications, modifying user dialogs, and recording and playing back tablet packets. Opening a manager handle requires a window handle. The window becomes a *manager window* and receives window messages about interface and context activity.

3.5. Extensions

The interface allows implementations to define additional features called *extensions*. Extensions can be made available to new applications without the need to modify existing applications. Extensions are supported through the information categories, through the flexible definition of packets, and through special context and manager functions.

Designing an extension involves defining the meaning and behavior of the extension packet and/or preference data, filling in the information category, defining the extension's interface with the special functions, and possibly defining additional functions to support the extension. Each extension will be assigned a unique *tag* for identification. Not all implementations will support all extensions.

A multiplexed information category contains descriptive data about extensions. Note that applications must find their extensions by iterating through the categories and matching tags. While tags are fixed across all implementations, category numbers may vary among implementations.

3.6. Persistent Binding of Interface Features (1.1)

The interface provides access to many of its features using consecutive numeric indices whose value is not guaranteed from session to session. However, sufficient information is provided to create unique identifiers for devices, cursors, and interface extensions. Devices should be uniquely identified by the contents of their name strings. If multiple identical devices are present, implementation providers should provide unique, persistent id strings to the extent possible. Identical devices that return unique serial numbers are ideal. If supported by the hardware, cursors also may have a *physical cursor id* that uniquely identifies the cursor in a persistent and stable manner. Interface extensions are uniquely identified by their tag.

4. Interface Implementations

Implementations of this interface usually support one specific device, a class of similar devices, or a common combination of devices. The following sections discuss guidelines for implementations.

4.1. File and Module Conventions

For 16-bit implementations, the interface functions, and any additional vendor- or device-specific functions, reside in a dynamic link library with the file name "WINTAB.DLL" and module name "WINTAB"; 32-bit implementations use the file name "WINTAB32.DLL" and module name "WINTAB32." Any other file or module conventions are implementation specific. Implementations may include other library modules or data files as necessary. Installation processes are likewise implementation-specific.

Wintab programs written in the C language require two header files. WINTAB.H contains definitions of all of the functions, constants, and fixed data types. PKTDEF.H contains a parameterized definition of the **PACKET** data structure, that can be tailored to fit the application. The Wintab Programmer's Kit contains these and other files necessary for Wintab programming, plus several example programs with C-language source files. The Wintab Programmer's Kit is available from the author.

4.2. Feature Support Options

Some features of the interface are optional and may be left out by some implementations.

Support of defined data items other than x, y, and buttons is optional. Many devices only report x, y, and button information.

Support of system-cursor contexts is optional. This option relieves implementations of replacing the system mouse driver in Windows versions before 3.1.

Support of Pen Windows contexts is optional. Not all systems will have the Pen Windows hardware and software necessary.

Support of external tablet manager applications is optional, and the number of manager handles is imple-mentation-dependent. However, the manager functions should be present in all implementations, returning appropriate failure codes if not fully implemented. An implementation may provide context- and hardware-management support internally only, if desired. On the other hand, providing the external manager interface may relieve the implementation of a considerable amount of user interface code, and make improvements to the manager interface easier to implement and distribute later.

Support of extension data items is optional. Most extensions will be geared to unusual hardware features.

5. Function Reference

All tablet function names have the prefix "WT" and have attributes equivalent to **WINAPI**. Applications gain access to the tablet interface functions through a dynamic-link library with standard file and module names, as defined in the previous section. Applications may link to the functions by using the Windows functions **LoadLibrary**, **FreeLibrary**, and **GetProcAddress**, or use an import library.

Specific to 32-bit Wintab: The functions **WTInfo, WTOpen, WTGet,** and **WTSet** have both ANSI and Unicode versions, using the same ANSI/Unicode porting conventions used in the Win32 API. Two non-portable functions, **WTQueuePackets, WTMgrCsrPressureBtnMarks**, are replaced by new portable functions **WTQueuePacketsEx, WTMgrCsrPressureBtnMarksEx.**

Table 0.1. Ordinal Function Numbers for Dynamic Linking

Ordinal numbers for dynamic linking are defined in the table below. Where two ordinal entries appear, the first entry identifies the 16-bit and 32-bit ANSI versions of the function. The second entry identifies the 32-bit Unicode version.

Function Name	Ordinal	Function Name	Ordinal
WTInfo	20, 1020	WTQueueSizeGet	84
WTOpen	21, 1021	WTQueueSizeSet	85
WTClose	22	WTMgrOpen	100
WTPacketsGet	23	WTMgrClose	101
WTPacket	24	WTMgrContextEnum	120
WTEnable	40	WTMgrContextOwner	121
WTOverlap	41	WTMgrDefContext	122
WTConfig	60	WTMgrDeviceConfig	140
WTGet	61, 1061	WTMgrExt	180
WTSet	62, 1062	WTMgrCsrEnable	181
WTExtGet	63	WTMgrCsrButtonMap	182
WTExtSet	64	WTMgrCsrPressureBtnMarks	183
WTSave	65	WTMgrCsrPressureResponse	184
WTRestore	66	WTMgrCsrExt	185
WTPacketsPeek	80	WTQueuePacketsEx	200

5.1. Basic Functions

The functions in the following section will be used by most tablet-aware applications. They include getting interface and device information, opening and closing contexts, and retrieving packets by polling or via Windows messages.

5.1.1. WTInfo

Syntax **UINT WTInfo(***wCategory, nIndex, lpOutput***)**

This function returns global information about the interface in an application-supplied buffer. Different types of information are specified by different index arguments. Applications use this function to receive information about tablet coordinates, physical dimensions, capabilities, and cursor types.

Parameter	**Type/Description**
wCategory	**UINT** Identifies the category from which information is being requested.
nIndex	**UINT** Identifies which information is being requested from within the category.
lpOutput	**LPVOID** Points to a buffer to hold the requested information.

Return Value The return value specifies the size of the returned information in bytes. If the information is not supported, the function returns zero. If a tablet is not physically present, this function always returns zero.

Comments Several important categories of information are available through this function. First, the function provides identification information, including specification and software version numbers, and tablet vendor and model information. Second, the function provides general capability information, including dimensions, resolutions, optional features, and cursor types. Third, the function provides categories that give defaults for all tablet context attributes. Finally, the function may provide any other implementation- or vendor-specific information categories necessary.

The information returned by this function is subject to change during a Windows session. Applications cannot change the information returned here, but tablet manager applications or hardware changes or errors can. Applications can respond to information changes by fielding the WT_INFOCHANGE message. The parameters of the message indicate which information has changed.

If the *wCategory* argument is zero, the function copies no data to the output buffer, but returns the size in bytes of the buffer necessary to hold the largest complete category. If the *nIndex* argument is zero, the function returns all of the information entries in the category in a single data structure.

If the *lpOutput* argument is NULL, the function just returns the required buffer size.

See Also Category and index definitions in tables 0.3 through 0.9, and the WT_INFOCHANGE message in section 0.0.0.

5.1.2. WTOpen

Syntax **HCTX WTOpen(***hWnd, lpLogCtx, fEnable***)**

This function establishes an active context on the tablet. On successful completion of this function, the application may begin receiving tablet events via messages (if they were requested), and may use the handle returned to poll the context, or to perform other context-related functions.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that owns the tablet context, and receives messages from the context.
lpLogCtx	**LPLOGCONTEXT** Points to an application-provided **LOGCONTEXT** data structure describing the context to be opened.
fEnable	**BOOL** Specifies whether the new context will immediately begin processing input data.

Return Value The return value identifies the new context. It is NULL if the context is not opened.

Comments Opening a new context allows the application to receive tablet input or creates a context that controls the system cursor or Pen Windows pen. The owning window (and all manager windows) will immediately receive a WT_CTXOPEN message when the context has been opened.

If the *fEnable* argument is zero, the context will be created, but will not process input. The context can be enabled using the **WTEnable** function.

If tablet event messages were requested in the context specification, the owning window will receive them. The application can control the message numbers used the **lcMsgBase** field of the **LOGCONTEXT** structure.

The window that owns the new context will receive context and information change messages even if event messages were not requested. It is not necessary to handle these in many cases, but some applications may wish to do so.

The newly opened tablet context will be placed on the top of the context overlap or-der.

Invalid or out-of-range attribute values in the logical context structure will either be validated, or cause the open to fail, depending on the attributes involved. Upon a successful return from the function, the context specification pointed to by *lpLogCtx* will contain the validated values.

See Also The **WTEnable** function in section 0.0.0, the **LOGCONTEXT** data structure in section 0.0.0, and the context and information change messages in sections 0.0 and 0.0.

5.1.3. WTClose

Syntax **BOOL WTClose(***hCtx***)**

This function closes and destroys the tablet context object.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context to be closed.

Return Value The function returns a non-zero value if the context was valid and was destroyed. Otherwise, it returns zero.

Comments After a call to this function, the passed handle is no longer valid. The owning window (and all manager windows) will receive a WT_CTXCLOSE message when the context has been closed.

See Also The **WTOpen** function in section **Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.

5.1.4. WTPacketsGet

Syntax	**int WTPacketsGet(***hCtx, cMaxPkts, lpPkts***)**

This function copies the next *cMaxPkts* events from the packet queue of context *hCtx* to the passed *lpPkts* buffer and removes them from the queue.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context whose packets are being returned.
cMaxPkts	**int** Specifies the maximum number of packets to return.
lpPkts	**LPVOID** Points to a buffer to receive the event packets.

Return Value The return value is the number of packets copied in the buffer.

Comments The exact structure of the returned packet is determined by the packet information that was requested when the context was opened.

The buffer pointed to by *lpPkts* must be at least *cMaxPkts* * sizeof(**PACKET**) bytes long to prevent overflow.

Applications may flush packets from the queue by calling this function with a NULL *lpPkt* argument.

See Also The **WTPacketsPeek** function in section 0.0.0, and the descriptions of the **LOGCONTEXT** (section 0.0.0) and **PACKET** (section 0.0.0) data structures.

5.1.5. WTPacket

Syntax **BOOL WTPacket(***hCtx, wSerial, lpPkt***)**

This function fills in the passed *lpPkt* buffer with the context event packet having the specified serial number. The returned packet and any older packets are removed from the context's internal queue.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context whose packets are being returned.
wSerial	**UINT** Serial number of the tablet event to return.
lpPkt	**LPVOID** Points to a buffer to receive the event packet.

Return Value The return value is non-zero if the specified packet was found and returned. It is zero if the specified packet was not found in the queue.

Comments The exact structure of the returned packet is determined by the packet information that was requested when the context was opened.

The buffer pointed to by *lpPkts* must be at least sizeof(**PACKET**) bytes long to prevent overflow.

Applications may flush packets from the queue by calling this function with a NULL *lpPkts* argument.

See Also The descriptions of the **LOGCONTEXT** (section 0.0.0) and **PACKET** (section 0.0.0) data structures.

5.2. Visibility Functions

The functions in this section allow applications to control contexts' visibility, whether or not they are processing input, and their overlap order.

5.2.1. WTEnable

Syntax **BOOL WTEnable(***hCtx, fEnable***)**

This function enables or disables a tablet context, temporarily turning on or off the processing of packets.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context to be enabled or disabled.
fEnable	**BOOL** Specifies enabling if non-zero, disabling if zero.

Return Value The function returns a non-zero value if the enable or disable request was satisfied, zero otherwise.

Comments Calls to this function to enable an already enabled context, or to disable an already disabled context will return a non-zero value, but otherwise do nothing.

The context’s packet queue is flushed on disable.

Applications can determine whether a context is currently enabled by using the **WTGet** function and examining the **lcStatus** field of the **LOGCONTEXT** structure.

See Also The **WTGet** function in section 0.0.0, and the **LOGCONTEXT** structure in section 0.0.0.

5.2.2. WTOverlap

Syntax **BOOL WTOverlap(***hCtx, fToTop***)**

This function sends a tablet context to the top or bottom of the order of overlapping tablet contexts.

Parameter	Type/Description
hCtx	**HCTX** Identifies the context to move within the overlap order.
fToTop	**BOOL** Specifies sending the context to the top of the overlap order if non-zero, or to the bottom if zero.

Return Value The function returns non-zero if successful, zero otherwise.

Comments Tablet contexts' input areas are allowed to overlap. The tablet interface maintains an overlap order that helps determine which context will process a given event. The topmost context in the overlap order whose input context encompasses the event, and whose event masks select the event will process the event.

This function is useful for getting access to input events when the application's context is overlapped by other contexts.

The function will fail only if the context argument is invalid.

5.3. Context Editing Functions

This group of functions allows applications to edit, save, and restore contexts.

5.3.1. WTGet

Syntax **BOOL WTGet(***hCtx, lpLogCtx***)**

This function fills the passed structure with the current context attributes for the passed handle.

Parameter	Type/Description
hCtx	**HCTX** Identifies the context whose attributes are to be copied.
lpLogCtx	**LPLOGCONTEXT** Points to a **LOGCONTEXT** data structure to which the context attributes are to be copied.

Return Value The function returns a non-zero value if the data is retrieved successfully. Otherwise, it returns zero.

See Also The **LOGCONTEXT** structure in section 0.0.0.

5.3.2. WTSet (1.1 modified)

Syntax **BOOL WTSet(***hCtx, lpLogCtx***)**

This function allows some of the context's attributes to be changed on the fly.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context whose attributes are being changed.
lpLogCtx	**LPLOGCONTEXT** Points to a **LOGCONTEXT** data structure containing the new context attributes.

Return Value The function returns a non-zero value if the context was changed to match the passed context specification; it returns zero if any of the requested changes could not be made.

Comments If this function is called by the task or process that owns the context, any context attribute may be changed. Otherwise, the function can change attributes that do not affect the format or meaning of the context's event packets and that were not speci-fied as locked when the context was opened. Context lock values can only be changed by the context's owner.

1.1: If the *hCtx* argument is a default context handle returned from **WTMgrDef-Context** or **WTMgrDefContextEx**, and the *lpLogCtx* argument is **WTP_LPDEFAULT**, the default context will be reset to its initial factory default values.

See Also The **LOGCONTEXT** structure in section 0.0.0 and the context lock values in table 0.13.

5.3.3. WTExtGet

Syntax **BOOL WTExtGet(***hCtx, wExt, lpData***)**

This function retrieves any context-specific data for an extension.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context whose extension attributes are being retrieved.
wExt	**UINT** Identifies the extension tag for which context-specific data is being retrieved.
lpData	**LPVOID** Points to a buffer to hold the retrieved data.

Return Value The function returns a non-zero value if the data is retrieved successfully. Otherwise, it returns zero.

See Also The extension definitions in Appendix B.

5.3.4. WTExtSet

Syntax **BOOL WTExtSet(***hCtx, wExt, lpData***)**

This function sets any context-specific data for an extension.

Parameter	Type/Description
hCtx	**HCTX** Identifies the context whose extension attributes are being modified.
wExt	**UINT** Identifies the extension tag for which context-specific data is being modified.
lpData	**LPVOID** Points to the new data.

Return Value The function returns a non-zero value if the data is modified successfully. Otherwise, it returns zero.

Comments Extensions may forbid their context-specific data to be changed during the lifetime of a context. For such extensions, calls to this function would always fail.

Extensions may also limit context data editing to the task of the owning window, as with the context locks.

See Also The extension definitions in Appendix B, the **LOGCONTEXT** data structure in section 0.0.0 and the context locking values in table 0.13.

5.3.5. WTSave

Syntax **BOOL WTSave(***hCtx, lpSaveInfo***)**

This function fills the passed buffer with binary save information that can be used to restore the equivalent context in a subsequent Windows session.

Parameter	Type/Description
hCtx	**HCTX** Identifies the context that is being saved.
lpSaveInfo	**LPVOID** Points to a buffer to contain the save information.

Return Value The function returns non-zero if the save information is successfully retrieved. Otherwise, it returns zero.

Comments The size of the save information buffer can be determined by calling the **WTInfo** function with category WTI_INTERFACE, index IFC_CTXSAVESIZE.

The save information is returned in a private binary data format. Applications should store the information unmodified and recreate the context by passing the save information to the **WTRestore** function.

Using **WTSave** and **WTRestore** allows applications to easily save and restore extension data bound to contexts.

See Also The **WTRestore** function in section 0.0.0.

5.3.6. WTRestore

Syntax	**HCTX WTRestore(***hWnd, lpSaveInfo, fEnable***)**

This function creates a tablet context from save information returned from the **WTSave** function.

<u>**Parameter**</u>	<u>**Type/Description**</u>
hWnd	**HWND** Identifies the window that owns the tablet context, and receives messages from the context.
lpSaveInfo	**LPVOID** Points to a buffer containing save information.
fEnable	**BOOL** Specifies whether the new context will immediately begin processing input data.

Return Value The function returns a valid context handle if successful. If a context equivalent to the save information could not be created, the function returns NULL.

Comments The save information is in a private binary data format. Applications should only pass save information retrieved by the **WTSave** function.

This function is much like **WTOpen**, except that it uses save information for input instead of a logical context. In particular, it will generate a WT_CTXOPEN message for the new context.

See Also The **WTOpen** function in section **Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**, the **WTSave** function in section 0.0.0, and the WT_CTXOPEN message in section 0.0.0.

5.4. Advanced Packet and Queue Functions

These functions provide advanced packet retrieval and queue manipulation. The packet retrieval functions require the application to provide a packet output buffer. To prevent overflow, the buffer must be large enough to hold the requested number of packets from the specified context. It is up to the caller to determine the packet size (by interrogating the context, if necessary), and to allocate a large enough buffer. Applications may flush packets from the queue by passing a NULL buffer pointer.

5.4.1. WTPacketsPeek

Syntax	**int WTPacketsPeek(***hCtx, cMaxPkts, lpPkts***)**

This function copies the next *cMaxPkts* events from the packet queue of context *hCtx* to the passed *lpPkts* buffer without removing them from the queue.

<u>**Parameter**</u>	<u>**Type/Description**</u>
hCtx	**HCTX** Identifies the context whose packets are being read.

cMaxPkts	**int** Specifies the maximum number of packets to return.
lpPkts	**LPVOID** Points to a buffer to receive the event packets.

Return Value The return value is the number of packets copied in the buffer.

Comments The buffer pointed to by *lpPkts* must be at least *cMaxPkts* * sizeof(**PACKET**) bytes long to prevent overflow.

See Also the WTPacketsGet function in section **Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.

5.4.2. WTDataGet

Syntax **int WTDataGet(***hCtx, wBegin, wEnd, cMaxPkts, lpPkts, lpNPkts***)**

This function copies all packets with serial numbers between *wBegin* and *wEnd* inclusive from the context's queue to the passed buffer and removes them from the queue.

Parameter	Type/Description
hCtx	**HCTX** Identifies the context whose packets are being returned.
wBegin	**UINT** Serial number of the oldest tablet event to return.
wEnd	**UINT** Serial number of the newest tablet event to return.
cMaxPkts	**int** Specifies the maximum number of packets to return.
lpPkts	**LPVOID** Points to a buffer to receive the event packets.
lpNPkts	**LPINT** Points to an integer to receive the number of packets actually copied.

Return Value The return value is the total number of packets found in the queue between *wBegin* and *wEnd*.

Comments The buffer pointed to by *lpPkts* must be at least *cMaxPkts* * sizeof(**PACKET**) bytes long to prevent overflow.

See Also The **WTDataPeek** function in section 0.0.0, and the **WTQueuePacketsEx** function in section 0.0.0.

5.4.3. WTDataPeek

Syntax **int WTDataPeek(***hCtx, wBegin, wEnd, cMaxPkts, lpPkts, lpNPkts***)**

This function copies all packets with serial numbers between *wBegin* and *wEnd* inclusive, from the context's queue to the passed buffer without removing them from the queue.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context whose packets are being read.
wBegin	**UINT** Serial number of the oldest tablet event to return.
wEnd	**UINT** Serial number of the newest tablet event to return.
cMaxPkts	**int** Specifies the maximum number of packets to return.
lpPkts	**LPVOID** Points to a buffer to receive the event packets.
lpNPkts	**LPINT** Points to an integer to receive the number of packets actually copied.

Return Value The return value is the total number of packets found in the queue between *wBegin* and *wEnd*.

Comments The buffer pointed to by *lpPkts* must be at least *cMaxPkts* * sizeof(**PACKET**) bytes long to prevent overflow.

See Also The **WTDataGet** function in section 0.0.0, and the **WTQueuePacketsEx** function in section 0.0.0.

5.4.4. WTQueuePackets (16-bit only)

Syntax **DWORD WTQueuePackets(***hCtx***)**

This function returns the serial numbers of the oldest and newest packets currently in the queue.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context whose queue is being queried.

Return Value The high word of the return value contains the newest packet's serial number; the low word contains the oldest.

Comments This function is non-portable and is superseded by **WTQueuePacketsEx.**

See Also The **WTQueuePacketsEx** function in section 0.0.0.

5.4.5. WTQueuePacketsEx

Syntax **BOOL WTQueuePacketsEx(***hCtx, lpOld, lpNew***)**

This function returns the serial numbers of the oldest and newest packets currently in the queue.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context whose queue is being queried.
lpOld	**UINT FAR *** Points to an unsigned integer to receive the oldest packet's serial number.
lpNew	**UINT FAR *** Points to an unsigned integer to receive the newest packet's serial number.

Return Value The function returns non-zero if successful, zero otherwise.

5.4.6. WTQueueSizeGet

Syntax **int WTQueueSizeGet(***hCtx***)**

This function returns the number of packets the context's queue can hold.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context whose queue size is being returned.

Return Value The return value is the number of packet the queue can hold.

See Also The **WTQueueSizeSet** function in section 0.0.0.

5.4.7. WTQueueSizeSet

Syntax **BOOL WTQueueSizeSet(***hCtx, nPkts***)**

This function attempts to change the context's queue size to the value specified in *nPkts*.

Parameter	**Type/Description**
hCtx	**HCTX** Identifies the context whose queue size is being set.
nPkts	**int** Specifies the requested queue size.

Return Value The return value is non-zero if the queue size was successfully changed. Otherwise, it is zero.

Comments If the return value is zero, the context has no queue because the function deletes the original queue before attempting to create a new one. The application must continue calling the function with a smaller queue size until the function returns a non-zero value.

See Also The **WTQueueSizeGet** function in section 0.0.0.

5.5. Manager Handle Functions

The functions described in this and subsequent sections are for use by tablet manager applications. The functions of this section create and destroy manager handles. These handles allow the interface code to limit the degree of simultaneous access to the powerful manager functions. Also, opening a manager handle lets the application receive messages about tablet interface activity.

5.5.1. WTMgrOpen

Syntax **HMGR WTMgrOpen(***hWnd, wMsgBase***)**

This function opens a tablet manager handle for use by tablet manager and configuration applications. This handle is required to call the tablet management functions.

<u>Parameter</u>	**<u>Type/Description</u>**
hWnd	**HWND** Identifies the window which owns the manager handle.
wMsgBase	**UINT** Specifies the message base number to use when notifying the manager window.

Return Value The function returns a manager handle if successful, otherwise it returns NULL.

Comments While the manager handle is open, the manager window will receive context messages from all tablet contexts. Manager windows also receive information change messages.

The number of manager handles available is interface implementation-dependent, and can be determined by calling the **WTInfo** function with category WTI_INTERFACE and index IFC_NMANAGERS.

See Also The **WTInfo** function in section **Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**, the **WTMgrClose** function in section 0.0.0, the description of message base numbers in section 0 and the context and information change messages in sections 0.0 and 0.0.

5.5.2. WTMgrClose

Syntax **BOOL WTMgrClose(***hMgr***)**

This function closes a tablet manager handle. After this function returns, the passed manager handle is no longer valid.

<u>Parameter</u>	**<u>Type/Description</u>**
hMgr	**HMGR** Identifies the manager handle to close.

Return Value The function returns non-zero if the handle was valid; otherwise, it returns zero.

5.6. Manager Context Functions

These functions provide access to all open contexts and their owners, and allow changing context defaults. Only tablet managers are allowed to manipulate tablet contexts belonging to other applications.

5.6.1. WTMgrContextEnum

Syntax **BOOL WTMgrContextEnum(***hMgr, lpEnumFunc, lParam***)**

This function enumerates all tablet context handles by passing the handle of each context, in turn, to the callback function pointed to by the *lpEnumFunc* parameter.

The enumeration terminates when the callback function returns zero.

Parameter	Type/Description
hMgr	**HMGR** Is the valid manager handle that identifies the caller as a manager application.
lpEnumFunc	**WTENUMPROC** Is the procedure-instance address of the call-back function. See the following "Comments" section for details.
lParam	**LPARAM** Specifies the value to be passed to the callback func-tion for the application's use.

Return Value The return value specifies the outcome of the function. It is non-zero if all contexts have been enumerated. Otherwise, it is zero.

Comments The address passed as the *lpEnumFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must have attributes equivalent to **WINAPI**. The callback function must have the following form:

Callback **BOOL WINAPI** *EnumFunc(hCtx, lParam)*
HCTX *hCtx;*
LPARAM *lParam;*

EnumFunc is a place holder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the applica-tion's module-definition file.

Parameter	Description
hCtx	Identifies the context.
lParam	Specifies the 32-bit argument of the **WTMgrContextEnum** func-tion.

Return Value

The function must return a non-zero value to continue enumeration, or zero to stop it.

5.6.2. WTMgrContextOwner

Syntax **HWND WTMgrContextOwner(***hMgr, hCtx***)**

This function returns the handle of the window that owns a tablet context.

Parameter	Type/Description
hMgr	**HMGR** Is the valid manager handle that identifies the caller as a manager application.
hCtx	**HCTX** Identifies the context whose owner is to be returned.

Return Value The function returns the context owner's window handle if the passed arguments are valid. Otherwise, it returns NULL.

Comments This function allows the tablet manager to coordinate tablet context management with the states of the context-owning windows.

5.6.3. WTMgrDefContext

Syntax **HCTX WTMgrDefContext(***hMgr, fSystem***) (1.4 modified)**

This function retrieves a context handle for either the default system context or the default digitizing context. This context is read-only.

Parameter	Type/Description
hMgr	**HMGR** Is the valid manager handle that identifies the caller as a manager application.
fSystem	**BOOL** Specifies retrieval of the default system context if non-zero, or the default digitizing context if zero.

Return Value The return value is the context handle for the specified default context, or NULL if the arguments were invalid.

Comments The default digitizing context is the context whose attributes are returned by the **WTInfo** function WTI_DEFCONTEXT category. The default system context is the context whose attributes are returned by the **WTInfo** function WTI_DEFSYSCTX category.

If any tablet is connected to the system, then the HCTX value returned by this function , when used with WTGet(), will return a LOGCONTEXT with lcDevice = (UINT)(-1), which represents the so-called "virtual device". A virtual device (or "virtual tablet") accepts data from any connected tablet and sends it to the application.

See Also The **WTInfo** function in section **Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.**Erreur ! Signet non défini.** the WTMgrDefContextEx function in section 0.0.0, and the category and index definitions in tables 0.3 through 0.9.

5.6.4. WTMgrDefContextEx (1.1 modified, 1.4 modified)

Syntax	**HCTX WTMgrDefContextEx(***hMgr, wDevice, fSystem***)**

This function retrieves a context handle that allows setting values for the default digitizing or system context for a specified device. This context is read-only.

Parameter	**Type/Description**
hMgr	**HMGR** Is the valid manager handle that identifies the caller as a manager application.
wDevice	**UINT** Specifies the device for which a default context handle will be returned.
fSystem	**BOOL** Specifies retrieval of the default system context if non-zero, or the default digitizing context if zero.

Return Value The return value is the context handle for the specified default context, or NULL if the arguments were invalid.

Comments The default digitizing contexts are contexts whose attributes are returned by the **WTInfo** function WTI_DDCTXS multiplexed category. The default system contexts are contexts whose attributes are returned by the **WTInfo** function WTI_DSCTXS multiplexed category.

If wDevice = (UINT)(-1), which represents the so-called "virtual device", then the HCTX value returned by this function , when used with WTGet(), will return a LOGCONTEXT with lcDevice = (UINT)(-1). A virtual device (or "virtual tablet") accepts data from any connected tablet and sends it to the application.

See Also The **WTInfo** function in section **Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**, and the category and index definitions in tables 0.3 through 0.9.

5.7. Manager Preference Data Functions (1.4 modified)

The functions in this section manipulate global ergonomic or preference settings that are not bound to tablet contexts.

5.7.1. WTMgrCsrButtonMap

Syntax **BOOL WTMgrCsrButtonMap(***hMgr, wCursor, lpLogBtns, lpSysBtns***)**

This function allows tablet managers to change the button mappings for each cursor type. Each cursor type has two button maps. The logical button map assigns physical buttons to logical buttons (applications only use logical buttons). The system cursor button map binds system-button functions to logical buttons.

Parameter	**Type/Description**
hMgr	**HMGR** Is the valid manager handle that identifies the caller as a manager application.
wCursor	**UINT** Specifies the zero-based cursor id of the cursor type whose button maps are being set.
lpLogBtns	**LPBYTE** Points to a 32 byte array of logical button numbers, one for each physical button. If the value **WTP_LPDEFAULT** is specified, the function resets the default value for the logical button map.
lpSysBtns	**LPBYTE** Points to a 32 byte array of button action codes, one for each logical button. If the value **WTP_LPDEFAULT** is specified, the function resets the default value for the system button map.

Return Value The return value is non-zero if the new settings have taken effect. Otherwise, it is zero.

Comments The function will allow two or more physical buttons to be assigned to one logical button. Likewise, it does not ensure that left and right system button events can be properly generated. It is up to the tablet manager to make sure that appropriate limitations are applied to the logical button map and that the system buttons are not left disabled.

See Also The cursor information categories in table 0.8, and the system button assignment codes in table 0.10.

5.7.2. WTMgrCsrPressureBtnMarks (16-bit only)

Syntax **BOOL WTMgrCsrPressureBtnMarks(***hMgr, wCsr, dwNMarks, dwTMarks***)**

This function allows tablet managers to change the pressure thresholds that control the mapping of pressure activity to button events.

Parameter	**Type/Description**
hMgr	**HMGR** Is the valid manager handle that identifies the caller as a manager application.
wCsr	**UINT** Specifies the zero-based cursor id of the cursor type whose pressure button marks are being set.
dwNMarks	**DWORD** Specifies the button marks for the normal pressure button. The low order word contains the release mark; the high order word contains the press mark. If the value **WTP_DWDEFAULT** is specified, the function resets the default value for the marks.

dwTMarks	**DWORD** Specifies the button marks for the tangent pressure button. The low order word contains the release mark; the high order word contains the press mark. If the value **WTP_DWDEFAULT** is specified, the function resets the default value for the marks.

Return Value The return value is non-zero if the new settings have taken effect. Otherwise, it is zero.

Comments This function is non-portable and is superseded by **WTMgrCsrPressureBtnMarksEx**.

See Also The cursor information categories in table 0.8, and the **WTMgrCsrPressureBtnMarksEx** function in section 0.0.0.

5.7.3. WTMgrCsrPressureBtnMarksEx

Syntax **BOOL WTMgrCsrPressureBtnMarksEx(***hMgr, wCsr, lpNMarks, lpTMarks***)**

This function allows tablet managers to change the pressure thresholds that control the mapping of pressure activity to button events.

<u>**Parameter**</u>	<u>**Type/Description**</u>
hMgr	**HMGR** Is the valid manager handle that identifies the caller as a manager application.
wCsr	**UINT** Specifies the zero-based cursor id of the cursor type whose pressure button marks are being set.
lpNMarks	**UINT FAR *** Points to a two-element array containing the button marks for the normal pressure button. The first unsigned integer contains the release mark; the second unsigned integer contains the press mark. If the value **WTP_LPDEFAULT** is specified, the function resets the default value for the marks.
lpTMarks	**UINT FAR *** Points to a two-element array containing the button marks for the tangent pressure button. The first unsigned integer contains the release mark; the second unsigned integer contains the press mark. If the value **WTP_LPDEFAULT** is specified, the function resets the default value for the marks.

Return Value The return value is non-zero if the new settings have taken effect. Otherwise, it is zero.

Comments This function allows the tablet manager to let the user control the "feel" of the pressure buttons when they are used as conventional buttons.

The pressure ranges are defined such that the minimum value of the range indicates the resting, unpressed state, and the maximum value indicates the fully pressed state. The button marks determine the pressure at which button press and release events are generated. When the button is logically up and pressure rises above the press mark, a button press event is generated. Conversely, when the button is logically down and pressure drops below the release mark, a button release event is generated.

The marks should be set far enough apart so that the button does not "jitter," yet far enough from the ends of the range so that the button events are easy to generate.

The marks are valid if and only if the button press mark is greater than the button release mark. Setting the marks to invalid combinations will turn off button-event generation for the button.

See Also The cursor information categories in table 0.8.

5.7.4. WTMgrCsrPressureResponse

Syntax **BOOL WTMgrCsrPressureResponse(***hMgr, wCsr, lpNResp, lpTResp***)**

This function allows tablet managers to tune the pressure response for each cursor type.

Parameter	**Type/Description**
hMgr	**HMGR** Is the valid manager handle that identifies the caller as a manager application.
wCsr	**UINT** Specifies the zero-based cursor id of the cursor whose pressure response curves are being set.
lpNResp	**UINT FAR *** Points to an array of UINTs describing the pressure response curve for normal pressure. If the value **WTP_LPDEFAULT** is specified, the function resets the default value for the response curve.
lpTResp	**UINT FAR *** Points to an array of UINTs describing the pressure response curve for tangential pressure. If the value **WTP_LPDEFAULT** is specified, the function resets the default value for the response curve.

Return Value The return value is non-zero if the new settings have taken effect. Otherwise, it is zero.

Comments The pressure response curves represent a transfer function mapping physical pressure values to logical pressure values (applications see only logical pressure values). Each array element contains a logical pressure value corresponding to some physical pressure value. The array indices are "stretched" evenly over the input range.

The number of entries in the arrays depends upon the range of physical pressure values. The number will be the number of physical values or 256, whichever is smaller. When there is an entry for each possible value, the value is used as an index to the array, and the corresponding array element is returned. When there are more values that array elements, the physical value is mapped to a 0-255 scale. The result is used as the index to look up the logical pressure value.

See Also The cursor information categories in table 0.8.

6. Message Reference

This section describes the messages sent to tablet-aware applications and to tablet managers. Since these messages are extensions to Windows, the message names below do not represent globally fixed message numbers. Applications choose a range of message numbers in the WM_USER range (WM_USER to 8000 hex), and publish their *message base* to the interface via the **lcMsgBase** field of the **LOGCONTEXT** structure (see section 0.0.0), or via the *wMsgBase* argument of the **WTMgrOpen** function (see section 0.0.0). Applications may choose different ranges for each context and manager handle, or use the same range for all.

The message names documented below actually represent fixed offsets from the application-selected base. The interface definition reserves sixteen offsets, to allow for future message definitions. The default message base is the global constant value WT_DEFBASE, defined as 7FF0 hex. Although this default value will be reasonably safe for many applications, it is the responsibility of application programmers to prevent message number conflicts.

Specific to 32-bit Wintab: Although most message arguments' types are wider for 32-bit Wintab, no message arguments have jumped from *wParam* to *lParam* or vice-versa. Most widened arguments now occupy previously unused or unavailable space. Please note that the category and index arguments of **WT_INFOCHANGE** have *not* been widened; they are still 16 bits. The header file WINTABX.H in the Wintab Programmer's Kit contains portable "message cracker" macros, suitable for both 16-bit and 32-bit use.

6.1. Event Messages

Wintab applications that receive device events via Windows messages will receive the WT_PACKET message.

6.1.1. WT_PACKET

Description The WT_PACKET message is posted to the context-owning windows that have requested messaging for their context.

Parameter	Description
wParam	Contains the serial number of the packet that generated the message.
lParam	Contains the handle of the context that processed the packet.

Comments Applications should call the **WTPacket** function or other packet retrieval functions upon receiving this message.

See Also The **WTPacket** function in section 0.0.0, and the advanced packet and queue functions in section 0.0.

6.1.2. WT_CSRCHANGE (1.1)

Description The WT_CSRCHANGE message is posted to the owning window when a new cursor enters the context.

Parameter	Description
wParam	Contains the serial number of the packet that generated the message.
lParam	Contains the handle of the context that processed the packet.

Comments Only contexts that have the CXO_CSRMESSAGES option selected will generate this message.

6.2. Context Messages

The messages described in this section provide information about changes in contexts.

6.2.1. WT_CTXOPEN

Description The WT_CTXOPEN message is sent to the owning window and to any manager windows when a context is opened.

Parameter	**Description**
wParam	Contains the context handle of the opened context.
lParam	Contains the current context status flags.

Comments Tablet manager applications should save the handle of the new context. Managers may want to query the user to configure the context further at this time.

6.2.2. WT_CTXCLOSE

Description The WT_CTXCLOSE message is sent to the owning window and to any manager windows when a context is about to be closed.

Parameter	**Description**
wParam	Contains the context handle of the context to be closed.
lParam	Contains the current context status flags.

Comments Tablet manager applications should note that the context is being closed and take appropriate action.

6.2.3. WT_CTXUPDATE

Description The WT_CTXUPDATE message is sent to the owning window and to any manager windows when a context is changed.

Parameter	**Description**
wParam	Contains the context handle of the changed context.
lParam	Contains the current context status flags.

Comments Applications may want to call **WTGet** or **WTExtGet** on receiving this message to find out what context attributes were changed.

6.2.4. WT_CTXOVERLAP

Description The WT_CTXOVERLAP message is sent to the owning window and to any manager windows when a context is moved in the overlap order.

Parameter	**Description**
wParam	Contains the context handle of the re-overlapped context.
lParam	Contains the current context status flags.

Comments Tablet managers can handle this message to keep track of context overlap requests by applications.

Applications can handle this message to find out when their context is obscured by another context.

6.2.5. WT_PROXIMITY

Description The WT_PROXIMITY message is posted to the owning window and any manager windows when the cursor enters or leaves context proximity.

Parameter	**Description**
wParam	Contains the handle of the context that the cursor is entering or leaving.
lParam	The low-order word is non-zero when the cursor is entering the context and zero when it is leaving the context. The high-order word is non-zero when the cursor is leaving or entering hardware proximity.

Comments Proximity events are handled separately from regular tablet events. Applications will receive proximity messages even if they haven't requested event messages.

6.3. Information Change Messages

The messages described in this section provide information about changes to information items.

6.3.1. WT_INFOCHANGE (1.4 modified)

Description The WT_INFOCHANGE message is sent to all manager and context-owning windows when the number of connected tablets has changed

Parameter	**Description**
wParam	Contains the manager handle of the tablet manager that changed the information, or zero if the change was reported through hardware.
lParam	Contains category and index numbers for the changed information. The low-order word contains the category number; the high-order word contains the index number.

Comments Applications can respond to capability changes by handling this message. Simple applications may want to prompt the user to close and restart them. The most basic applications that use the default context should not need to take any action.

Tablet managers should handle this message to react to hardware configuration changes or changes by other managers (in implementations that allow multiple tablet managers).

7. Data Reference

The data types in the following list are keywords that define the size and meaning of parameters and return values associated with tablet interface functions and messages. This list contains fixed-point arithmetic types, pointer types, bit field types, handles, and callback function prototypes. The fixed-point type names begin with the **FIX** prefix. The pointer-type names begin with either a **P** prefix (for short pointers) or an **LP** prefix (for long pointers). Bit field types contain commonly used sets of bits used to specify various conditions. The callback function prototypes use the **WT** prefix and the **PROC** suffix; they define the attributes, arguments and return types of application-defined callback functions. Handles provide access to resources managed internally by the tablet interface. The tablet interface data types are defined in the following list.

7.1. Common Data Types (1.1 modified, 1.4 modified)

The data types in the following list are keywords that define the size and meaning of parameters and return values associated with tablet interface functions and messages. This list contains fixed-point arithmetic types, pointer types, bit field types, handles, and callback function prototypes. The fixed-point type names begin with the **FIX** prefix. The pointer-type names begin with either a **P** prefix (for short pointers) or an **LP** prefix (for long pointers). Bit field types contain commonly used sets of bits used to specify various conditions. The callback function prototypes use the **WT** prefix and the **PROC** suffix; they define the attributes, arguments and return types of application-defined callback functions. Handles provide access to resources managed internally by the tablet interface. The tablet interface data types are defined in the following list.

Table 0.1. Common Data Types

Type	Definition
FIX32	A 32-bit fixed-point arithmetic type, with the radix point between the two words. Thus, the type contains 16 bits to the left of the radix point and 16 bits to the right of it.
HMGR	Handle for tablet manager. It is a value that allows applications to use tablet-management functions. When multiple manager applications are allowed, it also identifies the managers to the tablet interface.
HCTX	Handle to a tablet context. It is an index to the tablet interface's context tables.
LPLOGCONTEXT	Long pointer to a **LOGCONTEXT** data structure.
WTPKT	Bit field that specifies the various optional data items available in event packets. It is a 32-bit field. The event packet field flags can be combined using the bitwise OR operator. The **WTPKT** bit field can contain the values listed below, as well as any defined for extension data items.

Value	Meaning
PK_CONTEXT	Specifies the handle of the reporting context.
PK_STATUS	Specifies status information.
PK_TIME	Specifies the time at which the packet was generated.

PK_CHANGED	Specifies which packet data items have changed since the last packet.
PK_SERIAL_NUMBER	Specifies the packet serial number.
PK_CURSOR	Specifies the cursor that generated the packet.
PK_BUTTONS	Specifies packet button information.
PK_X, PK_Y, PK_Z	Specify packet x, y, and z axis data, respectively.
PK_NORMAL_PRESSURE, PK_TANGENT_PRESSURE	Specify tip-button or normal-to-surface pressure data, and barrel-button or tangent-to-surface pressure data, respectively.
PK_ORIENTATION	Specifies cursor orientation information.
PK_ROTATION (**1.1**)	Specifies cursor rotation information.

WTENUMPROC Specifies the prototype of a context enumeration callback:

BOOL (WINAPI * WTENUMPROC)(HCTX, LPARAM);

WTCONFIGPROC Specifies the prototype of a context configuration callback:

BOOL (WINAPI * WTCONFIGPROC)(HCTX, HWND);

7.2. Information Data Structures

This section describes the data formats returned by the **WTInfo** function and the category and index values used when calling the function. Some of the common data return formats are described first, followed by a complete listing of all the category and index values.

7.2.1. AXIS

Range and Resolution Descriptor

Description The **AXIS** data structure defines the range and resolution for many of the packet data items.

```
typedef struct tagAXIS {
        LONG    axMin;
        LONG    axMax;
        UINT    axUnits;
        FIX32   axResolution;

} AXIS;
```

The **AXIS** data structure has the following fields:

Field	**Description**
axMin	Specifies the minimum value of the data item in the tablet's native coordinates.
axMax	Specifies the maximum value of the data item in the tablet's native coordinates.
axUnits	Indicates the units used in calculating the resolution for the data item.
axResolution	Is a fixed-point number giving the number of data item increments per physical unit.

Comments This structure is used to construct return data for the **WTInfo** function. In particular, it is used within the WTI_DEVICES category for returning the capabilities of the device with respect to each axis or packet data item.

See Also The **WTInfo** function in section **Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.**Erreur ! Signet non défini.** and the values used for specifying units in table 0.2.

Table 0.2. Physical Unit Specifiers

TU_NONE	Specifies that no resolution in terms of physical units is given.
TU_INCHES	Specifies that resolution is given with respect to inches.
TU_CENTIMETERS	Specifies that resolution is given with respect to centimeters.
TU_CIRCLE	Specifies that resolution is given with respect to one full revolution of arc. For example, if a data item returns degrees, the resolution would be 360 and the units would be TU_CIRCLE. If the item were in radians, the resolution would be 6.28318 (to FIX32's precision) and the units would be TU_CIRCLE.

7.2.2. Information Categories and Indices (1.1 modified)

The categories, indices, and returned data formats are listed below. The WTI_DEFCONTEXT, WTI_DEFSYSCTX, WTI_DDCTXS, and WTI_DSCTXS categories share the same set of indices; they refer to the default digitizing and system contexts, respectively. The WTI_DEVICES, WTI_CURSORS and WTI_EXTENSIONS categories are multiplexed; the category names actually refer to the first of several consecutive categories.

Table 0.3. Category Definitions

Category	Description
WTI_INTERFACE	Contains global interface identification and capability information.
WTI_STATUS	Contains current interface resource usage statistics.
WTI_DEFCONTEXT	Contains the current default digitizing logical context.
WTI_DEFSYSCTX	Contains the current default system logical context.
WTI_DEVICES	Each contains capability and status information for a device.
WTI_CURSORS	Each contains capability and status information for a cursor type.
WTI_EXTENSIONS	Each contains descriptive information and defaults for an extension.
WTI_DDCTXS (**1.1**)	Each contains the current default digitizing logical context for the corresponding device.
WTI_DSCTXS (**1.1**)	Each contains the current default system logical context for the corresponding device.

Table 0.4. WTI_INTERFACE Index Definitions

Index	Type/Description
IFC_WINTABID	**TCHAR[]** Returns a copy of the null-terminated tablet hardware identification string in the user buffer. This string should include make, model, and revision information in user-readable format.
IFC_SPECVERSION	**WORD** Returns the specification version number. The high-order byte contains the major version number; the low-order byte contains the minor version number.
IFC_IMPLVERSION	**WORD** Returns the implementation version number. The high-order byte contains the major version number; the low-order byte contains the minor version number.
IFC_NDEVICES	**UINT** Returns the number of devices supported.
IFC_NCURSORS	**UINT** Returns the total number of cursor types supported.
IFC_NCONTEXTS	**UINT** Returns the number of contexts supported.
IFC_CTXOPTIONS	**UINT** Returns flags indicating which context options are supported (see table 0.11).
IFC_CTXSAVESIZE	**UINT** Returns the size of the save information returned from **WTSave**.

IFC_NEXTENSIONS	**UINT** Returns the number of extension data items supported.
IFC_NMANAGERS	**UINT** Returns the number of manager handles supported.

Table 0.5. WTI_STATUS Index Definitions

Index	Type/Description
STA_CONTEXTS	**UINT** Returns the number of contexts currently open.
STA_SYSCTXS	**UINT** Returns the number of system contexts currently open.
STA_PKTRATE	**UINT** Returns the maximum packet report rate currently being received by any context, in Hertz.
STA_PKTDATA	**WTPKT** Returns a mask indicating which packet data items are requested by at least one context.
STA_MANAGERS	**UINT** Returns the number of manager handles currently open.
STA_SYSTEM	**BOOL** Returns a non-zero value if system pointing is available to the whole screen; zero otherwise.
STA_BUTTONUSE	**DWORD** Returns a button mask indicating the logical buttons whose events are requested by at least one context.
STA_SYSBTNUSE	**DWORD** Returns a button mask indicating which logical buttons are assigned a system button function by the current cursor's system button map.

Table 0.6. WTI_DEFCONTEXT, WTI_DEFSYSCTX, WTI_DDCTXS (1.1), and WTI_DSCTXS (1.1) Index Definitions

Index	Type/Description
CTX_NAME	**TCHAR[]** Returns a 40 character array containing the default name. The name may occupy zero to 39 characters; the remainder of the array is padded with zeroes.
CTX_OPTIONS	**UINT** Returns option flags. For the default digitizing context, CXO_MARGIN and CXO_MGNINSIDE are allowed. For the default system context, CXO_SYSTEM is required; CXO_PEN, CXO_MARGIN, and CXO_MGNINSIDE are allowed. See table 0.11 for more information.
CTX_STATUS	**UINT** Returns zero.
CTX_LOCKS	**UINT** Returns which attributes of the default context are locked. See table 0.13 for more information.
CTX_MSGBASE	**UINT** Returns the value WT_DEFBASE. See the message descriptions in section 0.
CTX_DEVICE	**UINT** Returns the default device. If this value is -1, then it also known as a "virtual device".

CTX_PKTRATE	**UINT** Returns the default context packet report rate, in Hertz.
CTX_PKTDATA	**WTPKT** Returns which optional data items will be in packets returned from the context. For the default digitizing context, this field must at least indicate buttons, x, and y data.
CTX_PKTMODE	**WTPKT** Returns whether the packet data items will be returned in absolute or relative mode.
CTX_MOVEMASK	**WTPKT** Returns which packet data items can generate motion events in the context.
CTX_BTNDNMASK	**DWORD** Returns the buttons for which button press events will be processed in the context. The default context must at least select button press events for one button.
CTX_BTNUPMASK	**DWORD** Returns the buttons for which button release events will be processed in the context.
CTX_INORGX, CTX_INORGY, CTX_INORGZ	**LONG** Each returns the origin of the context's input area in the tablet's native coordinates, along the x, y, and z axes, respectively.
CTX_INEXTX, CTX_INEXTY, CTX_INEXTZ	**LONG** Each returns the extent of the context's input area in the tablet's native coordinates, along the x, y, and z axes, respectively.
CTX_OUTORGX, CTX_OUTORGY, CTX_OUTORGZ	**LONG** Each returns the origin of the context's output coordinate space in context output coordinates, along the x, y, and z axes, respectively.
CTX_OUTEXTX, CTX_OUTEXTY, CTX_OUTEXTZ	**LONG** Each returns the extent of the context's output coordinate space in context output coordinates, along the x, y, and z axes, respectively.
CTX_SENSX, CTX_SENSY, CTX_SENSZ	**FIX32** Each returns the relative-mode sensitivity factor, along the x, y, and z axes, respectively.
CTX_SYSMODE	**BOOL** Returns the default system cursor tracking mode.
CTX_SYSORGX, CTX_SYSORGY	**int** Each returns 0.
CTX_SYSEXTX, CTX_SYSEXTY	**int** Each returns the current screen display size in pixels, in the x and y directions, respectively.
CTX_SYSSENSX, CTX_SYSSENSY	**FIX32** Each returns the system cursor relative-mode sensitivity factor, in the x and y directions, respectively.

Table 0.7. WTI_DEVICES Index Definitions

Index	**Type/Description**
DVC_NAME	**TCHAR[]** Returns a displayable null- terminated string describing the device, manufacturer, and revision level.
DVC_HARDWARE	**UINT** Returns flags indicating hardware and driver capabilities, as defined below:
	Value / **Meaning**
	HWC_INTEGRATED: Indicates that the display and digitizer share the same surface.
	HWC_TOUCH: Indicates that the cursor must be in physical contact with the device to report position.
	HWC_HARDPROX: Indicates that device can generate events when the cursor is entering and leaving the physical detection range.
	HWC_PHYSID_CURSORS (**1.1**): Indicates that device can uniquely identify the active cursor in hardware.
DVC_NCSRTYPES	**UINT** Returns the number of supported cursor types.
DVC_FIRSTCSR	**UINT** Returns the first cursor type number for the device.
DVC_PKTRATE	**UINT** Returns the maximum packet report rate in Hertz.
DVC_PKTDATA	**WTPKT** Returns a bit mask indicating which packet data items are always available.
DVC_PKTMODE	**WTPKT** Returns a bit mask indicating which packet data items are physically relative, i.e., items for which the hardware can only report change, not absolute measurement.
DVC_CSRDATA	**WTPKT** Returns a bit mask indicating which packet data items are only available when certain cursors are connected. The individual cursor descriptions must be consulted to determine which cursors return which data.
DVC_XMARGIN, DVC_YMARGIN, DVC_ZMARGIN	**int** Each returns the size of tablet context margins in tablet native coordinates, in the x, y, and z directions, respectively.
DVC_X, DVC_Y, DVC_Z	**AXIS** Each returns the tablet's range and resolution capabilities, in the x, y, and z axes, respectively.
DVC_NPRESSURE, DVC_TPRESSURE	**AXIS** Each returns the tablet's range and resolution capabilities, for the normal and tangential pressure inputs, respectively.

DVC_ORIENTATION	**AXIS[]** Returns a 3-element array describing the tablet's orientation range and resolution capabilities.
DVC_ROTATION (**1.1**)	**AXIS[]** Returns a 3-element array describing the tablet's rotation range and resolution capabilities.
DVC_PNPID (**1.1**)	**TCHAR[]** Returns a null-terminated string containing the device's Plug and Play ID.

Table 0.8. WTI_CURSORS Index Definitions

Index	**Type/Description**
CSR_NAME	**TCHAR[]** Returns a displayable zero-terminated string containing the name of the cursor.
CSR_ACTIVE	**BOOL** Returns whether the cursor is currently connected.
CSR_PKTDATA	**WTPKT** Returns a bit mask indicating the packet data items supported when this cursor is connected.
CSR_BUTTONS	**BYTE** Returns the number of buttons on this cursor.
CSR_BUTTONBITS	**BYTE** Returns the number of bits of raw button data returned by the hardware.
CSR_BTNNAMES	**TCHAR[]** Returns a list of zero-terminated strings containing the names of the cursor's buttons. The number of names in the list is the same as the number of buttons on the cursor. The names are separated by a single zero character; the list is terminated by two zero characters.
CSR_BUTTONMAP	**BYTE[]** Returns a 32 byte array of logical button numbers, one for each physical button.
CSR_SYSBTNMAP	**BYTE[]** Returns a 32 byte array of button action codes, one for each logical button.
CSR_NPBUTTON	**BYTE** Returns the physical button number of the button that is controlled by normal pressure.
CSR_NPBTNMARKS	**UINT[]** Returns an array of two UINTs, specifying the button marks for the normal pressure button. The first UINT contains the release mark; the second contains the press mark.
CSR_NPRESPONSE	**UINT[]** Returns an array of UINTs describing the pressure response curve for normal pressure.
CSR_TPBUTTON	**BYTE** Returns the physical button number of the button that is controlled by tangential pressure.
CSR_TPBTNMARKS	**UINT[]** Returns an array of two UINTs, specifying the button marks for the tangential pressure button. The first UINT contains the release mark; the second contains the press mark.

CSR_TPRESPONSE	**UINT[]** Returns an array of UINTs describing the pressure response curve for tangential pressure.
CSR_PHYSID (**1.1**)	**DWORD** Returns a manufacturer-specific physical identifier for the cursor. This value will distinguish the physical cursor from others on the same device. This physical identifier allows applications to bind functions to specific physical cursors, even if category numbers change and multiple, otherwise identical, physical cursors are present.
CSR_MODE (**1.1**)	**UINT** Returns the cursor mode number of this cursor type, if this cursor type has the CRC_MULTIMODE capability.
CSR_MINPKTDATA (**1.1**)	**UINT** Returns the minimum set of data available from a physical cursor in this cursor type, if this cursor type has the CRC_AGGREGATE capability.
CSR_MINBUTTONS (**1.1**)	**UINT** Returns the minimum number of buttons of physical cursors in the cursor type, if this cursor type has the CRC_AGGREGATE capability.
CSR_CAPABILITIES (**1.1**)	**UINT** Returns flags indicating cursor capabilities, as defined below:

Value	**Meaning**
CRC_MULTIMODE	Indicates this cursor type describes one of several modes of a single physical cursor. Consecutive cursor type categories describe the modes; the CSR_MODE data item gives the mode number of each cursor type.
CRC_AGGREGATE	Indicates this cursor type describes several physical cursors that cannot be distinguished by software.
CRC_INVERT	Indicates this cursor type describes the physical cursor in its inverted orientation; the previous consecutive cursor type category describes the normal orientation.

Table 0.9. WTI_EXTENSIONS Index Definitions

Index	**Type/Description**
EXT_NAME	**TCHAR[]** Returns a unique, null-terminated string describing the extension.
EXT_TAG	**UINT** Returns a unique identifier for the extension.
EXT_MASK	**WTPKT** Returns a mask that can be bitwise OR'ed with **WTPKT**-type variables to select the extension.

EXT_SIZE	**UINT[]** Returns an array of two UINTs specifying the extension's size within a packet (in bytes). The first is for absolute mode; the second is for relative mode.
EXT_AXES	**AXIS[]** Returns an array of axis descriptions, as needed for the extension.
EXT_DEFAULT	**BYTE[]** Returns the current global default data, as needed for the extension. This data is modified via the **WTMgrExt** function.
EXT_DEFCONTEXT, EXT_DEFSYSCTX	**BYTE[]** Each returns the current default context-specific data, as needed for the extension. The indices identify the digitizing- and system-context defaults, respectively.
EXT_CURSORS	**BYTE[]** Is the first of one or more consecutive indices, one per cursor type. Each returns the current default cursor-specific data, as need for the extension. This data is modified via the **WTMgrCsrExt** function.

Table 0.10. System Button Assignment Values

The values defined below indicate emulations of mouse button actions. Three different types of actions are defined for each mouse button: clicking, double-clicking, and dragging. Clicking means the button behaves exactly like the corresponding mouse button. Double-clicking means the pressing the button once emulates a double-click of the mouse button. Dragging means that pressing the button toggles the state of the emulated mouse button: one click to press, one to release.

Value	**Meaning**
SBN_NONE	no system button actions.
SBN_LCLICK, SBN_LDBLCLICK, SBN_LDRAG	left button click, double-click, and drag, respectively.
SBN_RCLICK, SBN_RDBLCLICK, SBN_RDRAG	right button click, double-click, and drag, respectively.
SBN_MCLICK, SBN_MDBLCLICK, SBN_MDRAG	middle button click, double-click, and drag, respectively.

In implementations that support Pen Windows, the values defined below assign pen-button actions to tablet cursor buttons. They may be used alone or may be combined (using the bitwise OR operator) with one of the values defined above.

Value	**Meaning**
SBN_PTCLICK, SBN_PTDBLCLICK, SBN_PTDRAG	pen tip click (tap), double-click (double-tap), and drag (stroke), respectively.

SBN_PNCLICK, SBN_PNDBLCLICK, SBN_PNDRAG	inverted pen tip click, double-click, and drag, respectively.
SBN_P1CLICK, SBN_P1DBLCLICK, SBN_P1DRAG	barrel button 1 click, double-click, and drag, respectively.
SBN_P2CLICK, SBN_P2DBLCLICK, SBN_P2DRAG	barrel button 2 click, double-click, and drag, respectively.
SBN_P3CLICK, SBN_P3DBLCLICK, SBN_P3DRAG	barrel button 3 click, double-click, and drag, respectively.

7.3. Context Data Structures

This section describes the LOGCONTEXT data structure, which is used when opening and manipulating contexts. This structure contains everything applications and tablet managers need to know about a context. To simplify context manipulations, applications may want to take advantage of the default context specification available via the WTInfo function.

7.3.1. LOGCONTEXT (1.1 modified)

The **LOGCONTEXT** data structure contains the attributes necessary to specify a tablet context.

```
#define LC_NAMELEN 40
typedef struct tagLOGCONTEXT {
      TCHAR       lcName[LC_NAMELEN];
      UINT        lcOptions;
      UINT        lcStatus;
      UINT        lcLocks;
      UINT        lcMsgBase;
      UINT        lcDevice;
      UINT        lcPktRate;
      WTPKT       lcPktData;
      WTPKT       lcPktMode;
      WTPKT       lcMoveMask;
      DWORD       lcBtnDnMask;
      DWORD       lcBtnUpMask;
      LONG        lcInOrgX;
      LONG        lcInOrgY;
      LONG        lcInOrgZ;
      LONG        lcInExtX;
      LONG        lcInExtY;
      LONG        lcInExtZ;
      LONG        lcOutOrgX;
      LONG        lcOutOrgY;
      LONG        lcOutOrgZ;
      LONG        lcOutExtX;
      LONG        lcOutExtY;
      LONG        lcOutExtZ;
      FIX32       lcSensX;
      FIX32       lcSensY;
      FIX32       lcSensZ;
      BOOL        lcSysMode;
      int         lcSysOrgX;
      int         lcSysOrgY;
      int         lcSysExtX;
      int         lcSysExtY;
      FIX32       lcSysSensX;
      FIX32       lcSysSensY;
} LOGCONTEXT;
```

Description The **LOGCONTEXT** data structure has the following fields:

<u>**Field**</u>	<u>**Description**</u>
lcName	Contains a zero-terminated context name string.
lcOptions	Specifies options for the context. These options can be combined by using the bitwise OR operator. The **lcOptions** field can be any combination of the values defined in table 0.11. Specifying options that are unsupported in a particular implementation will cause **WTOpen** to fail.
lcStatus	Specifies current status conditions for the context. These conditions can be combined by using the bitwise OR operator. The **lcStatus** field can be any combination of the values defined in table 0.12.
lcLocks	Specifies which attributes of the context the application wishes to be locked. Lock conditions specify attributes of the context that cannot be changed once the context has been opened (calls to **WTConfig** will have no effect on the locked attributes). The lock conditions can be combined by using the bitwise OR operator. The **lcLocks** field can be any combination of the values defined in table 0.13. Locks can only be changed by the task or process that owns the context.
lcMsgBase	Specifies the range of message numbers that will be used for reporting the activity of the context. See the message descriptions in section 0.
lcDevice	Specifies the device whose input the context processes.
lcPktRate	Specifies the desired packet report rate in Hertz. Once the context is opened, this field will contain the actual report rate.
lcPktData	Specifies which optional data items will be in packets returned from the context. Requesting unsupported data items will cause **WTOpen** to fail.
lcPktMode	Specifies whether the packet data items will be returned in absolute or relative mode. If the item's bit is set in this field, the item will be returned in relative mode. Bits in this field for items not selected in the **lcPktData** field will be ignored. Bits for data items that only allow one mode (such as the serial number) will also be ignored.
lcMoveMask	Specifies which packet data items can generate move events in the context. Bits for items that are not part of the packet definition in the **lcPktData** field will be ignored. The bits for buttons, time stamp, and the serial number will also be ignored. In the case of overlapping contexts, movement events for data items not selected in this field may be processed by underlying contexts.

lcBtnDnMask	Specifies the buttons for which button press events will be processed in the context. In the case of overlapping contexts, button press events for buttons that are not selected in this field may be processed by underlying contexts.
lcBtnUpMask	Specifies the buttons for which button release events will be processed in the context. In the case of overlapping contexts, button release events for buttons that are not selected in this field may be processed by underlying contexts. If both press and release events are selected for a button (see the **lcBtnDnMask** field above), then the interface will cause the context to implicitly capture all tablet events while the button is down. In this case, events occurring outside the context will be clipped to the context and processed as if they had occurred in the context. When the button is released, the context will receive the button release event, and then event processing will return to normal.
lcInOrgX, lcInOrgY, lcInOrgZ	Each specifies the origin of the context's input area in the tablet's native coordinates, along the x, y, and z axes, respectively. Each will be clipped to the tablet native coordinate space when the context is opened or modified.
lcInExtX, lcInExtY, lcInExtZ	Each specifies the extent of the context's input area in the tablet's native coordinates, along the x, y, and z axes, respectively. Each will be clipped to the tablet native coordinate space when the context is opened or modified.
lcOutOrgX, lcOutOrgY, lcOutOrgZ	Each specifies the origin of the context's output area in context output coordinates, along the x, y, and z axes, respectively. Each is used in coordinate scaling for absolute mode only.
lcOutExtX, lcOutExtY, lcOutExtZ	Each specifies the extent of the context's output area in context output coordinates, along the x, y, and z axes, respectively. Each is used in coordinate scaling for absolute mode only.
lcSensX, lcSensY, lcSensZ	Each specifies the relative-mode sensitivity factor for the x, y, and z axes, respectively.
lcSysMode	Specifies the system cursor tracking mode. Zero specifies absolute; non-zero means relative.
lcSysOrgX, lcSysOrgY	Together specify the origin of the screen mapping area for system cursor tracking, in screen coordinates.
lcSysExtX, lcSysExtY	Together specify the extent of the screen mapping area for system cursor tracking, in screen coordinates.
lcSysSensX, lcSysSensY	Each specifies the system-cursor relative-mode sensitivity factor for the x and y axes, respectively.

Comments The **LOGCONTEXT** structure determines what events an application will get, how they will be processed, and how they will be delivered to the application or to Windows itself.

Tablet contexts reflect the tension between application control of input data and user control of workstation ergonomics. Some attributes of contexts are controlled by the user, others by the application. The notion of locks allows applications to control some of the attributes that are normally controlled by the user. The user can always control the origin of a tablet context's input area (its physical location on the tablet). The lockable attributes are controlled by either the user or the application, at the application's discretion. Everything else is controlled by the application. This arrangement allows programmers to control the complexity of their applications, while allowing users to arrange their workstations in a comfortable, efficient manner.

Tablet contexts control scaling of input points. The scaling works in a manner analogous to the scaling features provided in Windows' GDI interface. The tablet input area is mapped to the output coordinate space using transformation equations similar to the equations for the GDI's MM_ANISOTROPIC scaling mode. In other words, the entire input context is mapped to the entire output context. The scaling transformation can change the unit size, the origin or offset of the returned coordinates, and the direction of the axes, just as in GDI scaling.

System cursor scaling and tracking can be controlled independently from tablet packet scaling and tracking. Thus, system contexts may be use polling to receive auxiliary tablet data in any desired form. Absolute-mode system cursor contexts can be mapped to a sub-context of the screen, such as an application drawing window. This feature allows absolute mode data entry and scaling of drawings with existing, unmodified applications.

The exact scaling equations for each axis are as follows.

```
Let:

In = the input value for the axis.
Out = the output value for the axis
InOrg = the input origin (such as lcInOrgX) for the axis
InExt = the input extent (such as lcInExtX) for the axis
OutOrg = the output origin (such as lcOutOrgX or lcSysOrgX) for
        the axis
OutExt = the output extent (such as lcOutExtX or lcSysExtX) for
        the axis
sign() returns 1 if the input value is greater than or equal to
        0, -1 otherwise.
abs() returns the absolute value of its input.

if sign(OutExt) == sign(InExt)
        Out = ((In - InOrg) * abs(OutExt) / abs(InExt)) + OutOrg
else
        Out = ((abs(InExt) - (In - InOrg)) * abs(OutExt) /
                abs(InExt)) + OutOrg
```

The move mask and button masks together determine what kinds of events will be processed by the context.

See Also The context option values in table 0.11, the context status values in table 0.12, and the context lock values in table 0.13.

Table 0.11. Context Option Values

Value	Meaning
CXO_SYSTEM	Specifies that the context is a system cursor context.
CXO_PEN	Specifies that the context is a Pen Windows context, if Pen Windows is installed. The context is also a system cursor context; specifying CXO_PEN implies CXO_SYSTEM.
CXO_MESSAGES	Specifies that the context returns WT_PACKET messages to its owner.
CXO_MARGIN	Specifies that the input context on the tablet will have a margin. The margin is an area outside the specified input area where events will be mapped to the edge of the input area. This feature makes it easier to input points at the edge of the context.
CXO_MGNINSIDE	If the CXO_MARGIN bit is on, specifies that the margin will be inside the specified context. Thus, scaling will occur from a context slightly smaller than the specified input context to the output coordinate space.
CXO_CSRMESSAGES (**1.1**)	Specifies that the context returns WT_CSRCHANGE messages to it owner.

Table 0.12. Context Status Values

Value	Meaning
CXS_DISABLED	Specifies that the context has been disabled using the **WTEnable** function.
CXS_OBSCURED	Specifies that the context is at least partially obscured by an overlapping context that is higher in the context overlap order.
CXS_ONTOP	Specifies that the context is the topmost context in the context overlap order.

Table 0.13. Context Lock Condition Values

Value	Meaning
CXL_INSIZE	Specifies that the context's input size cannot be changed. When this value is not specified, the context's input extents in x, y, and z can be changed. NOTE: The context's origins in x, y, and z can always be changed.
CXL_INASPECT	Specifies that the context's input aspect ratio cannot be changed. When this value is specified, the context's size can be changed, but the ratios among x, y, and z extents will be kept as close to constant as possible.
CXL_MARGIN	Specifies that the context's margin options cannot be changed. This value controls the locking of the CXO_MARGIN and CXO_MGNINSIDE option values.

CXL_SENSITIVITY	Specifies that the context's sensitivity settings for x, y, and z cannot be changed.
CXL_SYSOUT	If the context is a system cursor context, the value specifies that the system pointing control variables of the context cannot be changed.

7.4. Event Data Structures

This section describes the data structures used to return events. The definitions include the family of variable data structures for packets, and related structures and constants.

7.4.1. PACKET (1.1 modified)

Event Packet Data Structure

Description The **PACKET** data structure is a flexible structure that contains tablet event information. Each of its fields is optional. The structure consists of a concatenation of the data items selected in the **lcPktData** field of the context that generated the packet. The order of the data items is the same as the order of the corresponding set bits in the field.

The **pkButtons** data item has different formats in absolute and relative modes, as determined by the PK_BUTTONS bit in the **lcPktMode** field of the context.

The example structure below illustrates the case where all of the defined bits in the **lcPktData** field are set and the **lcPktMode** field is clear (all items are in absolute mode).

```
typedef struct tagPACKET {
      HCTX        pkContext;
      UINT        pkStatus;
      LONG        pkTime;
      WTPKT       pkChanged;
      UINT        pkSerialNumber;
      UINT        pkCursor;
      DWORD       pkButtons;
      DWORD       pkX;
      DWORD       pkY;
      DWORD       pkZ;
      UINT        pkNormalPressure;
      UINT        pkTangentPressure;
      ORIENTATION pkOrientation;
      ROTATION    pkRotation;  /* 1.1 */

} PACKET;
```

The **PACKET** data structure has the following defined fields:

Field	Description
pkContext	Specifies the context that generated the event.
pkStatus	Specifies various status and error conditions. These conditions can be combined by using the bitwise OR operator. The **pkStatus** field can be any combination of the values defined in table 0.14.
pkTime	In absolute mode, specifies the system time at which the event was posted. In relative mode, specifies the elapsed time in milliseconds since the last packet.
pkChanged	Specifies which of the included packet data items have changed since the previously posted event.
pkSerialNumber	Contains a serial number assigned to the packet by the context. Consecutive packets will have consecutive serial numbers.
pkCursor	Specifies which cursor type generated the packet.
pkButtons	In absolute mode, is a **DWORD** containing the current button state. In relative mode, is a **DWORD** whose low word contains a button number, and whose high word contains one of the following codes:

Value	Meaning
TBN_NONE	no change in button state.
TBN_UP	button was released.
TBN_DOWN	button was pressed.

Field	Description
pkX, pkY, pkZ	In absolute mode, each is a **DWORD** containing the scaled cursor location along the x, y, and z axes, respectively. In relative mode, each is a **LONG** containing the scaled change in cursor position.
pkNormalPressure, pkTangentPressure	In absolute mode, each is a **UINT** containing the adjusted state of the normal and tangent pressures, respectively. In relative mode, each is an **int** containing the change in adjusted pressure state.
pkOrientation	Contains updated cursor orientation information. For details, see the description of the **ORIENTATION** data structure in section 0.0.0.
pkRotation (1.1)	Contains updated cursor rotation information. For details, see the description of the **ROTATION** data structure in section 0.0.0.

Comments	Extension data items are also concatenated onto **PACKET** data structures, when supported by the implementation, and selected by using the appropriate mask in the **lcPktData** context field. Extension items are concatenated after the standard data items, in increasing order of their unique tags (using unsigned comparison). Within an implementation, extension masks will be assigned in tag order. While the value of an extension's mask is not guaranteed to be the same across implementations, the extension's tag (and thus the order of extensions within the **PACKET** structure) will always be the same.
See Also	The packet status values in table 0.14, the **ORIENTATION** structure in section 0.0.0, the **ROTATION** structure in section 0.0.0, the **WTOpen** function in section **Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**.**Erreur ! Signet non défini.**, **LOGCONTEXT** data structure in section 0.0.0, and the WTI_EXTENSIONS group of information categories in table 0.9.

Table 0.14. Packet Status Values

<u>**Value**</u>	<u>**Meaning**</u>
TPS_PROXIMITY	Specifies that the cursor is out of the context.
TPS_QUEUE_ERR	Specifies that the event queue for the context has overflowed.
TPS_MARGIN	Specifies that the cursor is in the margin of the context.
TPS_GRAB	Specifies that the cursor is out of the context, but that the context has grabbed input while waiting for a button release event.
TPS_INVERT (**1.1**)	Specifies that the cursor is in its inverted state.

7.4.2. ORIENTATION

Cursor Orientation Descriptor

Description The **ORIENTATION** data structure specifies the orientation of the cursor with respect to the tablet.

```
typedef struct tagORIENTATION {
      int   orAzimuth;
      int   orAltitude;
      int   orTwist;
} ORIENTATION;
```

The **ORIENTATION** data structure has the following fields:

<u>**Field**</u>	<u>**Description**</u>
orAzimuth	Specifies the clockwise rotation of the cursor about the z axis through a full circular range.

orAltitude	Specifies the angle with the x-y plane through a signed, semicircular range. Positive values specify an angle upward toward the positive z axis; negative values specify an angle downward toward the negative z axis.
orTwist	Specifies the clockwise rotation of the cursor about its own major axis.

Comments Each cursor type will have a major axis and "normal orientation" defined for it, based on its physical characteristics.

See Also The **PACKET** data structure definition in section 0.0.0.

7.4.3. ROTATION (1.1)

Cursor Rotation Descriptor

Description The **ROTATION** data structure specifies the Rotation of the cursor with respect to the tablet.

```
typedef struct tagROTATION {
      int   roPitch;
      int   roRoll;
      int   roYaw;
} ROTATION;
```

The **ROTATION** data structure has the following fields:

<u>**Field**</u>	<u>**Description**</u>
roPitch	Specifies the pitch of the cursor.
roRoll	Specifies the roll of the cursor.
roYaw	Specifies the yaw of the cursor.

Comments Each cursor type will have rotation semantics defined for it, based on its physical characteristics.

See Also The **PACKET** data structure definition in section 0.0.0.

Appendix A. Using PKTDEF.H

This appendix describes how to use the header file PKTDEF.H in C-language programs.

The program must first include WINTAB.H, since PKTDEF.H depends on definitions in WINTAB.H.

If the program will use just one packet format, the following steps are necessary before including PKTDEF.H. The program must define the PACKETDATA constant to indicate which packet data items to include, and the PACKETMODE constant to indicate which packet data items are in relative mode. Both constants should be combinations of **WTPKT** bits (the PK_* identifiers), combined with the bitwise OR operator. The generated structure typedef will be called PACKET. Use the PACKETDATA and PACKETMODE constants to fill in the **lcPktData** and **lcPktMode** fields of the **LOGCONTEXT** structure. See example #1 below.

Example #1.-- single packet format

```
#include <wintab.h>
#define PACKETDATA     PK_X | PK_Y | PK_BUTTONS      /* x, y, buttons */
#define PACKETMODE     PK_BUTTONS                    /* buttons relative mode */
#include <pktdef.h>
...
       lc.lcPktData = PACKETDATA;
       lc.lcPktMode = PACKETMODE;
```

If the program uses multiple packet formats, PKTDEF.H can generate multiple packet structures with different names, data items, and modes. The program will include PKTDEF.H once for each packet structure. Before each PKTDEF.H inclusion, define a constant PACKETNAME. Its text value will be a prefix for this packet's parameters and names. Next define <name>PACKETDATA and <name>PACKETMODE, where <name> is the *value* of the PACKETNAME constant just defined. Finally include PKTDEF.H to define a structure named <name>PACKET. See example #2 below.

Example #2. -- multiple formats

```
#include <wintab.h>
#define PACKETNAME            MOE
#define MOEPACKETDATA         PK_X | PK_Y | PK_BUTTONS      /* x, y, buttons */
#define MOEPACKETMODE         PK_BUTTONS                    /* buttons relative mode */
#include <pktdef.h>
#define PACKETNAME            LARRY
#define LARRYPACKETDATA       PK_Y | PK_Z | PK_BUTTONS      /* y, z, buttons */
#define LARRYPACKETMODE       PK_BUTTONS                    /* buttons relative mode */
#include <pktdef.h>
#define PACKETNAME            CURLY
#define CURLYPACKETDATA       PK_X | PK_Z | PK_BUTTONS      /* x, z, buttons */
#define CURLYPACKETMODE       PK_BUTTONS                    /* buttons relative mode */
#include <pktdef.h>
...
       lcMOE.lcPktData = MOEPACKETDATA;
       lcMOE.lcPktMode = MOEPACKETMODE;
...
       lcLARRY.lcPktData = LARRYPACKETDATA;
       lcLARRY.lcPktMode = LARRYPACKETMODE;
...
       lcCURLY.lcPktData = CURLYPACKETDATA;
       lcCURLY.lcPktMode = CURLYPACKETMODE;
```

Appendix B. Extension Definitions (1.4 modified)

This appendix describes in more detail the programming interface for extensions. It also includes definitions of all currently defined extensions.

Table B.1. Current Defined Extensions (1.4 modified)

Name	**Tag Constant Symbol**	**Tag Constant Value**
Out of Bounds Tracking	WTX_OBT	0
Cursor Mask	WTX_CSRMASK	3
Extended Button Masks	WTX_XBTNMASK	4
Touch Strips	WTX_TOUCHSTRIP	6
Touch Rings	WTX_TOUCHRING	7
Express Keys	WTX_EXPKEYS2	8

Note (1.4 modified) *- if a WTX_* value is not included in Table B.1, then it is not supported.*

B.1. Extensions Programming

Each extension has several parts to its interface. It has a unique identifying number called its tag. It has an information category, multiplexed under WTI_EXTENSIONS. Optionally, it may also define a packet data item, or respond to one or more of the functions **WTExtGet, WTExtSet, WTMgrExt,** or **WTMgrCsrExt.**

The header file WINTAB.H includes tag definitions for currently defined extensions. The tag identifiers have the prefix "WTX_".

Since not all Wintab implementations support all extensions, applications must detect extension support at run-time, using a function like the one shown below. The function takes an extension's tag, and returns its information category offset from WTI_EXTENSIONS.

```
UINT ScanExts(UINT wTag)
{
        UINT i;
        UINT wScanTag;

        /* scan for wTag's info category. */
        for (i = 0; WTInfo(WTI_EXTENSIONS + i, EXT_TAG, &wScanTag); i++) {
                 if (wTag == wScanTag) {
                        /* return category offset from WTI_EXTENSIONS. */
                        return i;
                }
        }
        /* return error code. */
        return 0xFFFF;
}
```

Extensions that have a packet data item will not have the same **WTPKT** bit assigned across implementations, so extension packet data items use a slightly different scheme for controlling PKTDEF.H. For each such extension, the application defines a constant PACKET<extension name> with a value of either PKEXT_ABSOLUTE or PKEXT_RELATIVE. In the following example, the application adds the packet data item for the extension "XFOO".

```
#include <wintab.h>
#define PACKETDATA     PK_X | PK_Y | PK_BUTTONS       /* x, y, buttons */
#define PACKETMODE     PK_BUTTONS                     /* buttons relative mode */
#define PACKETXFOO     PKEXT_ABSOLUTE                 /* XFOO absolute mode */
#include <pktdef.h>
```

The application should then detect if XFOO is supported at run-time, and retrieve its **WTPKT** bit mask from XFOO's EXT_MASK information item.

Extensions Programming Notes

NOTE: It is recommended that applications remove their overrides when the application's main window is no longer active (use WM_ACTIVATE). Extension overrides take effect across the entire system; if an application leaves its overrides in place, that control will not function correctly in other applications.

When an extension event occurs, an extension packet is created and queued to the context. If the application requested messages then a WT_PACKETEXT message is sent to the application. This message is identical to the WT_PACKET message except for the queue. The application should call the WTPacket function with the context provided and a PACKETEXT will be returned. This packet contains an EXTENSIONBASE data structure as well as any other structures the application requested at PKTDEF.h include time.

B.2. Out of Bounds Tracking

The Wintab Out of Bounds Tracking (OBT) extension enables cursor tracking in areas not covered by any active tablet context. Normally, when the cursor moves outside all active contexts, Wintab event activity ceases, until the cursor again enters some context. Wintab simply discards such out of bounds events. With OBT enabled, Wintab delivers out of bounds events to some context, usually the one the cursor most recently left. The receiving context clips the event into its coordinate range. The user usually sees the cursor moving along the edge of the screen, or some window, as when a mouse user tries to roll the screen cursor off the edge of the screen.

The OBT logic routes unclaimed events by several rules. Only contexts that capture motion events (i.e., their logical context **lcMoveMask** field has the PK_X, PK_Y, or PK_Z bits set) are eligible to receive out-of-bounds events. Whenever an eligible context processes an event, it becomes the current OBT context. When an out-of-bounds event takes place, the current OBT context, if any, receives it. There is no current OBT context at system startup, or when the current OBT context is closed or disabled. In such situations, an out-of-bounds event goes to the first eligible context in the context stacking order.

OBT is very similar to the margin feature controlled by the CXO_MARGIN and CXO_MGNINSIDE context options. In fact, to applications, out-of-bounds events are indistinguishable from margin events; both will have the TPS_MARGIN flag set in their packet's **pkStatus** field.

OBT Programming

The OBT extension supports enabling or disabling OBT behavior on a per-device basis. Any Wintab aware program can detect OBT support and read its current state. Wintab manager programs can turn OBT on or off for any or all devices. OBT uses the extension tag value zero. Updated versions of WINTAB.H define the constant WTX_OBT as zero.

Information Category

The OBT Info category implements only the EXT_NAME, EXT_TAG, and EXT_DEFAULT indices. Using the other the indices in the category will cause **WTInfo** to return zero. The size of the EXT_DEFAULT data depends on the number of currently installed Wintab devices. Applications should use either the **WTInfo** function's return value for EXT_DEFAULT, or the number of devices indicated in the IFC_NDEVICES information item to determine the appropriate buffer size.

Index	Value
EXT_NAME	**TCHAR[]** the null-terminated string "Out of Bounds Tracking"
EXT_TAG	**UINT** the tag value zero
EXT_DEFAULT	**BOOL[]** an array of flags, one per device, that are non-zero if OBT is enabled for the corresponding device

Turning OBT On and Off

A Wintab manager program can turn OBT on and off using the **WTMgrExt** function. The passed data buffer has the same format as the EXT_DEFAULT information item.

B.5. Cursor Mask

The Wintab Cursor Mask (CSRMASK) extension allows contexts to select input based on the cursor type. Such selection supports easy assignment of functionality to specific cursors, particularly on devices that support multiple active cursors.

CSRMASK Programming

There are three main steps to using CSRMASK: detecting CSRMASK support at run-time, creating a context, and modifying the context's cursor mask using the functions **WTExtGet** and **WTExtSet**. Applications supporting CSRMASK should use updated version of WINTAB.H with the tag constant WTX_CSRMASK (defined as 3). See also the example program CSRMASK in the Wintab Programmer's Kit.

Detect CSRMASK by scanning the WTI_EXTENSIONS information categories for one having the WTX_CSRMASK tag. If none is found, the CSRMASK extension is not supported and the application should not attempt to use CSRMASK. If CSRMASK is supported, the application should use the **WTOpen** function with the enable flag argument set to FALSE, set the desired CSRMASK value using the function **WTExtSet**, and finally enable the context with **WTEnable**.

The cursor mask data is 16-byte bitmap, in which each bit controls input selection for a corresponding cursor id. Bits 0 to 7 of byte 0 correspond to cursor ids 0 to 7, bits 0 to 7 of byte 1 correspond to cursor ids 8 to 15, and so on. A value of 1 in the bit means the context will process input from the cursor, a value of 0 means the context will ignore that cursor.

Note that in multiple-device configurations, the cursor ids of the device in question may not start at zero. The first cursor id of a given device is available from the DVC_FIRSTCSR item of the device information category. Also, note that cursor ids are not guaranteed to remain the same from session to session, if other devices or cursor types are installed or removed. Persistent functional bindings on cursor types should use the device name (DVC_NAME) and physical cursor id (CSR_PHYSID) to guarantee binding stability between sessions.

Information Category

The CSRMASK information category implements only the items listed below.

Index	Value
EXT_NAME	**TCHAR[]** the null-terminated string "Cursor Mask"
EXT_TAG	**UINT** the tag value three.
EXT_DEFCONTEXT	**BYTE[16]** the default cursor mask; all bits set to one.
EXT_DEFSYSCTX	**BYTE[16]** the default cursor mask; all bits set to one.

B.6. Extended Button Masks

The Wintab Extended Button Masks (XBTNMASK) extension allows contexts to fully select input of up to 256 buttons. This supports locator devices with great numbers of buttons.

XBTNMASK Programming

There are three main steps to using XBTNMASK: detecting XBTNMASK support at run-time, creating a context, and modifying the context's cursor mask using the functions **WTExtGet** and **WTExtSet**. Applications supporting XBTNMASK should use updated version of WINTAB.H with the tag constant WTX_XBTNMASK (defined as 4).

Detect XBTNMASK by scanning the WTI_EXTENSIONS information categories for one having the WTX_XBTNMASK tag. If none is found, the XBTNMASK extension is not supported and the application should not attempt to use XBTNMASK. If XBTNMASK is supported, the application should use the **WTOpen** function with the enable flag argument set to FALSE, set the desired XBTNMASK value using the function **WTExtSet**, and finally enable the context with **WTEnable**.

The button mask data structure consists of two 32-byte bitmaps, in which each bit controls input selection for a corresponding button. Bits 0 to 7 of byte 0 correspond to buttons 0 to 7, bits 0 to 7 of byte 1 correspond to buttons 8 to 15, and so on. A value of 1 in the bit means the context will process input from the button, a value of 0 means the context will ignore that button. The first 32-byte map controls button down events; the second controls button up events. The button mask data structure definition follows.

```
typedef struct tagXBTNMASK {
        BYTE xBtnDnMask[32];
        BYTE xBtnUpMask[32];
} XBTNMASK;
```

These extended button masks extend the button masks in the logical context structure, but their behavior is otherwise identical to that described for **lcBtnDnMask** and **lcBtnUpMask**. Conflicts between the values of the logical context masks and the extension masks will be resolved by combining the low four bytes of the masks via a bitwise AND operation. Both the extension masks and the logical context masks returned by subsequent **WTGet** calls will be updated.

Information Category

The XBTNMASK information category implements only the items listed below.

Index	Value
EXT_NAME	**TCHAR[]** the null-terminated string "Extended Button Mask"
EXT_TAG	**UINT** the tag value four.
EXT_DEFCONTEXT	**XBTNMASK** the default masks; all bits set to one.
EXT_DEFSYSCTX	**XBTNMASK** the default masks; all bits set to one.
EXT_CURSORS	**BYTE[512]** the system and logical button maps for each cursor type. The first 256 bytes comprise the system button map; the second 256 bytes contain the logical button map.

B.7. Express Keys Extension (1.4 modified)

Some tablets have physical key on the tablet surface. We have received requests to expose these keys to an application and allow the application to perform tasks when the key is pressed.

This extension allows an application to receive Express Key event notifications, and to override the normal system-wide behavior of Express Keys. For a working example, see the TabletControlsSample demo from the Windows download sample code at: http://www.wacomeng.com/windows/index.html

To make use of this extension, perform the following steps:

a. Make sure you have versions of WINTAB.H and PKTDEF.H that support this extension (≥ 1.4).

   The following information items are supported for this extension:

   - `EXT_NAME` — (`TCHAR[]`)
     The null-terminated string "`ExpressKeys`"
   - `EXT_TAG` — (`UINT`)
     The tag value `WTX_EXPKEYS2`
   - `EXT_MASK` — (`WTPKT`)
     The masked used when creating the context to activate the extension.
   - `EXT_SIZE` — (`UINT[2]`)
     The size of the data structure in the extension packet.

- `EXT_AXES` — (`AXIS[1]`)
  An axis structure with `axMin` equal to 0, `axMax` equal to the value returned if all button are pressed, `axResolution` is 1 and `axUnits` is `TU_NONE`.

b. Detect at run-time whether the Express Key extension is supported by the installed driver. This can be done by scanning the WTI_EXTENSIONS information categories for an extensions whose tag is equal to WTX_EXPKEYS2. If found, the extension is supported. (see PKTDEF.H for details)

```
// Iterate through Wintab extension indices
for ( UINT i=0, thisTag = 0;
      WTInfo(WTI_EXTENSIONS+i, EXT_TAG, &thisTag);
      i++)
{
    // looking for the specified tag
    if (thisTag == WTX_EXPKEYS2)
    {
        // note the index of the found tag
        return i;
    }
}
return ERROR("EXPKEYS2 not supported by this Wintab");
```

c. Add the Express Keys data to your extension packet definition by adding the following declaration before including PKTDEF.H:

#define PACKETEXPKEYS PKEXT_ABSOLUTE

This causes the EXPKEYSDATA structure to be included in the PACKETEXT structure definition. It is added as pkExpKeys.

Get the required mask for the extension and the mask with the lcPktData member of the context.

```
// get the extension mask for the express keys
WTInfo(WTI_EXTENSIONS + extIndex_ExpKeys, EXT_MASK,
&lExpKeys_Mask);

...

lcContext.lcPktData = PACKETDATA | lExpKeys_Mask;
```

This tells the driver to fill in the EXPKEYSDATA structure.

d. Determine the keys that are present by requesting extension data through the WTExtGet command:

```
// allocate a buffer
BYTE *buffer = new BYTE [sizeof(EXTPROPERTY) + sizeof(BOOL)];
// cast the buffer to the extension property data structure
EXTPROPERTY *prop = (EXTPROPERTY*)buffer;

// fill in the data
prop->version = 0;
```

```
prop->tabletIndex = tablet_I;
prop->controlIndex = control_I;
prop->functionIndex = function_I;
prop->propertyID = TABLET_PROPERTY_AVAILABLE;
prop->reserved = 0;
prop->dataSize = sizeof(BOOL);

// send the command to Wintab
if (WTExtGet(ghCtx, ext_I, prop))
{
    // store the data requested
    return *((BOOL*)(&prop->data[0]));
}
```

WTExtGet uses the EXTPROPERTY structure to recieve data about the extension.

e. Override ExpressKey behavior by calling WTExtSet.

```
// allocate a buffer
BYTE *buffer = new BYTE [sizeof(EXTPROPERTY) + sizeof(BOOL)];
// cast the buffer to the extension property data structure
EXTPROPERTY *prop = (EXTPROPERTY*)buffer;

// fill in the data
prop->version = 0;
prop->tabletIndex = tablet_I;
prop->controlIndex = control_I;
prop->functionIndex = function_I;
prop->propertyID = TABLET_PROPERTY_OVERRIDE;
prop->reserved = 0;
prop->dataSize = sizeof(BOOL);
*((BOOL*)(&prop->data[0])) = TRUE;

// send the command to Wintab
return !WTExtSet(ghCtx, ext_I, prop);
```

With WTExtSet command, you can also override the name of the Express Key. (see the WINTAB.H for a complete list of properties)

Express Key Display Icons

Some tablets have display icons associated with the Express Keys.

a. To determine if an express key has a display you need to check the TABLET_PROPERTY_ICON_FORMAT property for that express key.

```
// allocate a buffer
BYTE *buffer = new BYTE [sizeof(EXTPROPERTY) + sizeof(UINT32)];
// cast the buffer to the extension property data structure
EXTPROPERTY *prop = (EXTPROPERTY*)buffer;

// fill in the data
prop->version = 0;
```

```
prop->tabletIndex = tablet_I;
prop->controlIndex = control_I;
prop->functionIndex = function_I;
prop->propertyID = TABLET_PROPERTY_ICON_FORMAT;
prop->reserved = 0;
prop->dataSize = sizeof(UINT32);

// send the command to Wintab
if (WTExtGet(ghCtx, ext_I, prop))
{
    // store the data requested
    return *((UINT32*)(&prop->data[0]));
}
```

If the return value is TABLET_ICON_FMT_NONE the express key has no display.

b. To change the image on the display first override the key and then send an override icon to the display.

```
// allocate a buffer
BYTE *buffer = new BYTE [sizeof(EXTPROPERTY) + imageSize];
// cast the buffer to the extension property data structure
EXTPROPERTY *prop = (EXTPROPERTY*)buffer;

// fill in the data
prop->version = 0;
prop->tabletIndex = tablet_I;
prop->controlIndex = control_I;
prop->functionIndex = function_I;
prop->propertyID = TABLET_PROPERTY_OVERRIDE_ICON;
prop->reserved = 0;
prop->dataSize = imageSize;
for (int i = 0; i < imageSize; i++)
{
    prop->data[i] = imageData[i];
}

// send the command to Wintab
return !WTExtSet(ghCtx, ext_I, prop);
```

The image data should follow be in BMP or PNG format.

B.8. Touch Strips and Touch Rings Extensions (1.4 modified)

These extensions allow an application to receive Touch Strip and Touch Ring event notifications, and to override the normal system-wide behavior of Touch Strip and Touch Ring. To make use of this extension, perform the following steps:

a. Make sure you have versions of WINTAB.H and PKTDEF.H that support this extension (≥ 1.4).

The following information items are supported for the touch strip extension:

- EXT_NAME — (TCHAR[])
  The null-terminated string "TouchStrips"
- EXT_TAG — (UINT)
  The tag value WTX_TOUCHSTRIP
- EXT_MASK — (WTPKT)
  The masked used when creating the context to activate the extension.
- EXT_SIZE — (UINT[2])
  The size of the data structure in the extension packet.
- EXT_AXES — (AXIS[1])
  An axis structure with axMin equal to 0, axMax equal to the value returned if all button are pressed, axResolution is 1 and axUnits is TU_NONE.

b. The following information items are supported for the touch ring extension:

- EXT_NAME — (TCHAR[])
  The null-terminated string "TouchRings"
- EXT_TAG — (UINT)
  The tag value WTX_TOUCHRING
- EXT_MASK — (WTPKT)
  The masked used when creating the context to activate the extension.
- EXT_SIZE — (UINT[2])
  The size of the data structure in the extension packet.
- EXT_AXES — (AXIS[1])
  An axis structure with axMin equal to 0, axMax equal to the number of zones on the touch ring, axResolution is 1/axMax and axUnits is TU_CIRCLE.

c. Detect at run-time whether the extension is supported by the installed driver. This can be done by scanning the WTI_EXTENSIONS information categories for an extensions whose tag is equal to WTX_TOUCHSTRIP and/or WTX_TOUCHRING respectfully. If found, the extension is supported. (see PKTDEF.H for details)

```
// Iterate through Wintab extension indices
for ( UINT i=0, thisTag = 0;
      WTInfo(WTI_EXTENSIONS+i, EXT_TAG, &thisTag);
      i++)
{
    // looking for the specified tag
    if (thisTag == WTX_TOUCHSTRIP)
    {
        // note the index of the found tag
        return i;
    }
}
return ERROR("TOUCHSTRIP not supported by this Wintab");
```

d. Add the Express Keys data to your extension packet definition by adding the following declaration before including PKTDEF.H:

```
#define PACKETTOUCHSTRIP PKEXT_ABSOLUTE

#define PACKETTOUCHRING PKEXT_ABSOLUTE
```

This causes the SLIDERDATA structure to be included in the PACKETEXT structure definition. The touchstrip extension adds pkTouchStrip; while the touchring extension adds pkTouchRing.

e. Get the required mask for the extension and add the mask with the lcPktData member of the context.

```
// get the extension mask for the touch strip
WTInfo(WTI_EXTENSIONS + extIndex_TouchStrip, EXT_MASK,
&lTouchStrip_Mask);

// get the extension mask for the touch ring
WTInfo(WTI_EXTENSIONS + extIndex_TouchRing, EXT_MASK,
&lTouchRing_Mask);

...

lcContext.lcPktData = PACKETDATA | lTouchStrip_Mask |
lTouchRing_Mask;
```

This tells the driver to fill in the correct SLIDERDATA structure.

f. Determine the functions that are present by requesting extension data through the WTExtGet command:

```
// allocate a buffer
BYTE *buffer = new BYTE [sizeof(EXTPROPERTY) + sizeof(BOOL)];
// cast the buffer to the extension property data structure
EXTPROPERTY *prop = (EXTPROPERTY*)buffer;

// fill in the data
prop->version = 0;
prop->tabletIndex = tablet_I;
prop->controlIndex = control_I;
prop->functionIndex = function_I;
prop->propertyID = TABLET_PROPERTY_AVAILABLE;
prop->reserved = 0;
prop->dataSize = sizeof(BOOL);

// send the command to Wintab
if (WTExtGet(ghCtx, ext_I, prop))
{
    // store the data requested
    return *((BOOL*)(&prop->data[0]));
}
```

WTExtGet uses the EXTPROPERTY structure to recieve data about the extension.

g. Override the function behavior by calling WTExtSet.

```
// allocate a buffer
BYTE *buffer = new BYTE [sizeof(EXTPROPERTY) + sizeof(BOOL)];
// cast the buffer to the extension property data structure
EXTPROPERTY *prop = (EXTPROPERTY*)buffer;
```

```
// fill in the data
prop->version = 0;
prop->tabletIndex = tablet_I;
prop->controlIndex = control_I;
prop->functionIndex = function_I;
prop->propertyID = TABLET_PROPERTY_OVERRIDE;
prop->reserved = 0;
prop->dataSize = sizeof(BOOL);
*((BOOL*)(&prop->data[0])) = TRUE;

// send the command to Wintab
return !WTExtSet(ghCtx, ext_I, prop);
```

With WTExtSet command, you can also override the name of the function. (see the WINTAB.H for a complete list of properties)

B.9. Extension Structures (1.4 modified)

EXPKEYSDATA structure

```
typedef struct tagEXPKEYSDATA { /* 1.4 */
    BYTE    nTablet;    // index of the tablet
    BYTE    nControl;   // position index of the control
    BYTE    nLocation;  // location of the control
    BYTE    nReserved;  //
    DWORD   nState;     // state of the control (1 - pressed, 0 -
released)
} EXPKEYSDATA;
```

This structure is returned as part of the PACKETEXT structure.

SLIDERDATA structure

```
typedef struct tagSLIDERDATA { /* 1.4 */
    BYTE    nTablet;    // index of the tablet
    BYTE    nControl;   // position index of the control
    BYTE    nFunction;  // control function index
    BYTE    nReserved;  //
    DWORD   nPosition;  // position of the finger on the control (zero
for no touch)
} SLIDERDATA;
```

This structure is returned as part of the PACKETEXT structure.

EXTPROPERTY structure

```
typedef struct tagEXTPROPERTY { /* 1.4 */
    BYTE    version;        // Structure version, 0 for now
    BYTE    tabletIndex;    // 0-based index for tablet
    BYTE    controlIndex;   // 0-based index for control
    BYTE    functionIndex;  // 0-based index for control's sub-function
    WORD    propertyID;     // property ID
    WORD    reserved;       // DWORD-alignment filler
    DWORD   dataSize;       // number of bytes in data[] buffer
    BYTE    data[1];        // raw data
} EXTPROPERTY;
```

This structure is used for WTExtGet and WTExtSet.

Note that this stucture is of variable size. The total size of the structure is sizeof(EXTPROPERTY) + dataSize. Take care to allocate the correct amount of memory before using this structure.

APPENDIX C. ADDITIONAL FEATURES (1.4 MODIFIED)

C.1. 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.

There are several ways to detect the use of the eraser.

One way is to look for a negative orAltitude value in the pkOrientation data of a packet. If you use this method please note that the tilt value is defined as unsigned (although it is stated to be signed in the Wintab specification) so you will have to cast the tilt value as signed to make this work.

Another way to detect the eraser is to look for cursor number 2, and with the advent of the Dual Track (explained later) also check for cursor number 5. The time to check the cusor number is once the device is brought into proximity. The WT_CSRCHANGE message is will allow you to detect when a new device enters proximity. For backwards compatibility you might also process the WT_PROXIMITY message. Be sure to include the cursor number (pkCursor field) in the data packet. If you wish to use this method you should ask Wintab for the number of cursors per device (WTInfo(WTI_INTERFACE, IFC_NCURSORS, &data) and modulus that result with the pkCursor field to do the test.

The third way an application can detect the eraser is by looking at the TPS_INVERT bit-field in the pkStatus data item of a packet. This field was added to the Wintab 1.1 specification.

C.2. Tilt

Stylus tilt has several uses, but the most obvious is to change brush shape. Tilt can also be used for object rotation, or as a joystick.

During initialization you can check for the existence of tilt this way:

```
// Get info. about tilt
struct tagAXIS tiltOrient[3];
BOOL tiltSupport = FALSE;
tiltSupport = WTInfo(WTI_DEVICES, DVC_ORIENTATION, &tiltOrient);
if (tiltSupport )
{
    // Does the tablet support azimuth and altitude
    if (tiltOrient[0].axResolution && tiltOrient[1].axResolution)
    {
        // Get resolution
```

```
            azimuth = tiltOrient[0].axResolution;
            altitude = tiltOrient[1].axResolution;
        }
        else
        {
            tiltSupport = FALSE;
        }
    }
```

Tilt data is in the pkOrientation packet.

C.3. Dual Track

Dual Track, 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 certain supported Intuos tablets it is possible to track ANY two devices.

Note: Dual track was only supported for the following tablets. UDII 12x12, Intuos and Intuos II. Intuos III and IV do not support Dual track.

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*, by 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 dual track:

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.

Pointing devices are indexed from 0 to 5. The application can tell which device generated a packet by looking at the packet’s pkCursor data item. The possible index values are:

Index 0 – Puck-like device #1

Index 1 – Stylus-like device #1

Index 2 – Inverted stylus-like device #1

Index 3 – Puck-like device #2

Index 4 – Stylus-like device #2

Index 5 – Inverted stylus-like device #2

The indexes are really placeholders for actual pointing devices. For example, the first time device index 2 enters the proximity of a context, it may be an inverted airbrush. As long as device index 2 is in proximity of the context, the device index of 2 uniquely identifies this physical device as the same inverted airbrush. However, after device 2 exits proximity, the next time the context sees a pointing device with an index of 2 enter proximity, index 2 may identify a different device, such as an inverted pen.

By default, tracking is turned on for the first device (pkCursor = 0, 1, or 2). This is the first device to enter proximity. Tracking is turned off (pkCursor = 3, 4, or 5) for the second device to enter proximity. So, an application that does not support dual 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 a WT_CSRCHANGE message) based on which device is in which hand.

Use the cursor number (pkCursor field) in the data packet to separate the packet stream into two streams (0-2 and 3-5) one steam for each device.

C.4. Unique ID

There is another new feature to the Intuos series: Unique ID. A chip with a unique number is inside every stylus device so every device can be uniquely identified. With Unique ID you can assign a specific drawing tool to a specific pointing 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 CSR_TYPE, is actually a code unique to each device type. The other section, the CSR_PHYSID, is a unique 32 bit number within a device type. CSR_PHYSID may be repeated between device types, but not within a type.

The CSR_TYPE 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.

Bits 1,2,8,9,10,11 identify the device. The middle 4 bits (4,5,6,7) 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 one side switch. Bits 0,3 are used internally and should be ignored.

The currently supported types are:

General stylus: (CSR_TYPE & 0x0F06) == 0x0802

Airbrush: (CSR_TYPE & 0x0F06) == 0x0902

Art Pen: (CSR_TYPE & 0x0F06) == 0x0804

4D Mouse: (CSR_TYPE & 0x0F06) == 0x0004

5 button puck: (CSR_TYPE & 0x0F06) == 0x0006

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

C.5. Airbrush Fingerwheel

The Intuos series tablets have a device that is similar to an airbrush. The device 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 to vary ink density and, using pressure to vary line width.

The value of the fingerwheel on the side of the airbrush is reported in the pkTangentPressure (tangential pressure) field so be sure to include it in your packets. The value varies between 0 and 1023. But don’t hard code 1024! Use the function WTInfo(WTI_DEVICES, DVC_TPRESSURE) to query the tangent pressure range supported by the tablet.

C.6. 4D Mouse Rotation

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

4D Mouse axial rotation is reported in the pkOrientation.orTwist field. Values in this field range from 0 to 3599.

During initialization you can check for the existence of rotation this way:

```
// Get info. about rotation
struct tagAXIS tiltOrient[3];
BOOL rotateSupport = FALSE;
rotateSupport = WTInfo(WTI_DEVICES, DVC_ORIENTATION, &tiltOrient);
if (rotateSupport)
{
    // Does the tablet support twist
    if (!tiltOrient[2].axResolution)
    {
        rotateSupport = FALSE;
    }
}
```

You would probably check for this at the same time you check for tilt, since it is in the same data structure.

C.7. 4D Mouse Thumbwheel

Uses for the 4D Mouse thumbwheel include zooming and 3D navigation. The thumbwheel value is reported in the pkZ field. The thumbwheel varies from pkZ = 1023 to pkZ = -1023 and snaps to the middle when released. The middle is 0, and the total range is from axMin to axMax. The value axResolution is 0 if the tablet does not support the 4D Mouse (such as legacy tablets). 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.

During initialization you can check for the existence of the thumbwheel this way:

```
// Get info. about thumbwheel
struct tagAXIS thumbwheelAxis;
BOOL thumbwheelSupport = FALSE;
thumbwheelSupport = WTInfo(WTI_DEVICES, DVC_Z, &thumbwheelAxis);
if (thumbWheelSupport)
{
    // Does the tablet support thumbwheel?
    if (!thumbwheelAxis.axResolution)
    {
        thumbwheelSupport = FALSE;
    }
}
```