Ink Serialized Format Specification

Table of Contents

Introduction

This document describes the Windows Ink Services Platform's (WISP) Ink Serialized Format (ISF). The intent of this document is:

- To describe how WISP Ink data is serialized into a stream. This document does not intend to describe a file format. Applications will have their own file format and will store serialized Ink (WISPINK) as a separate stream within their own files. However a single WISPINK stream may be stored in a file for those applications that do not have a file format or need to store ink in individual files. The file extension in this case should be .WNK.

- To explain to a developer reading this specification how to write an interoperable application that uses ISF.

- To enable someone to examine a stream and using this specification determine if it is a valid ISF stream.

This document is organized into several sections. The "Overview" section discusses the goals of the ISF and provides general background information. The "General Description" section explains ISF and provides examples, discusses critical details such as the low level encoding, and explains the common components found in an ISF stream. However, as with any format specification, to fully describe the detailed semantics, interactions and relationships between the components requires a Bacus-Naur Form (BNF) description. This document provides a BNF specification of ISF for those who need very precise details about the Ink format. The final section is an "Appendix" that lists other details necessary to implement ISF.

Overview

WISP is a set of COM services that an application can use to capture, manipulate, and store INK. One of these services enables an application to read and write ink using the WISP ISF.

In the simplest case, ink is simply a sequence of strokes, where each stroke is comprised of a sequence of points, and the points are X, and Y coordinates. Many of the new devices can provide information such as pressure, and angle, while new applications need to be able to store custom information along with the ink data.

General Description

ISF has the following basic structure shown in the diagram below. Each ISF stream begins with a number that identifies the version of ISF used when writing the file. Following the ISF Version number is a number that indicates the size, in bytes, of all the Global Ink Properties and all the Strokes. Next comes a list of all the properties, both custom and predefined, that apply to the ink as a whole and last comes the list of strokes that make up the ink.

Figure **Error! Bookmark not defined.**- ISF Basic Structure

Each of these items will be discussed in more detail later. However the key thing to note is that predefined properties, custom properties, and even strokes are all of the same form, with some minor exceptions. The basic form for these properties is what we call a “Tagged” structure.

A “Tagged” structure, as shown below, begins with an identifying “tag” followed by a “size field” followed by data. The “tag” identifies the contents of the data while the “size field” identifies the size of the data in bytes. The tag may be either a predefined tag or an application-specific custom tag. Predefined tags are listed at the end in the appendix.

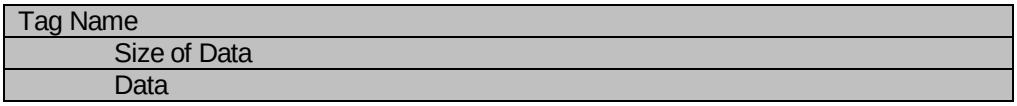

Figure **Error! Bookmark not defined.** - Tagged Structure

Both the “List of Global Ink Properties” and “List of Strokes” are a sequence of these tagged structures. A sequence with two tagged structures is show below:

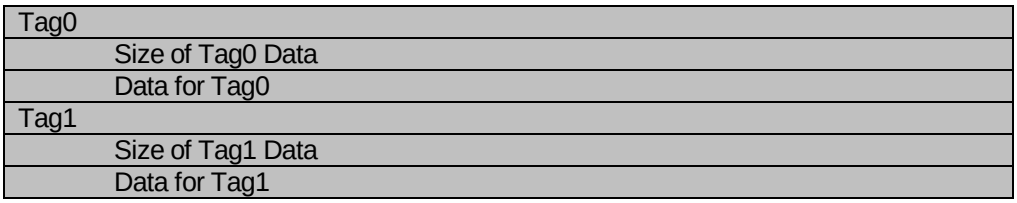

Figure **Error! Bookmark not defined.** - Sequence of two tagged structures.

Furthermore the “Data” may, in turn, contain more tagged structures and so on. The tagged structure can be a recursive hierarchical data structure. An expanded example is:

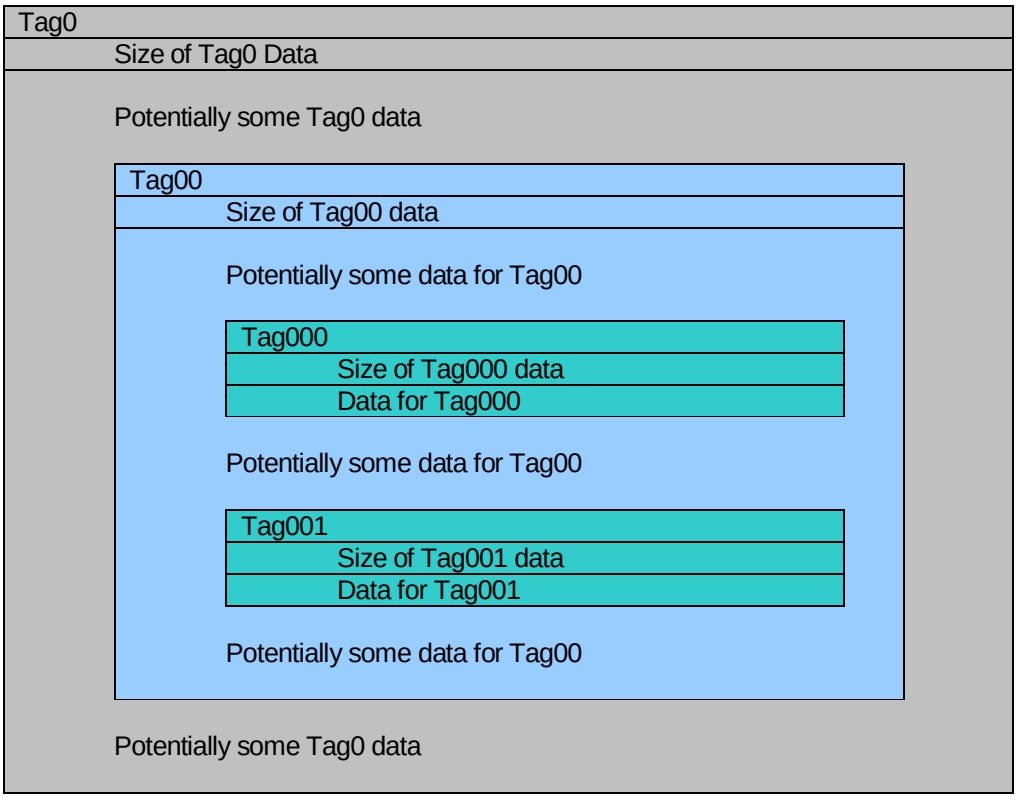

Figure **Error! Bookmark not defined.** - Hierarchical Tagged Structure

The “tag” describing the “data” determines whether the “data” contains tagged structures or even how many tagged structures. For example, custom properties are considered opaque to ISF since they are application-defined and thus only contain data. However, a stroke may have one or more stroke properties, which are also represented as tagged structures.

In ISF, the tags are “represented” by GUIDS. Using GUIDS guarantees that the tags will always be unique for all vendors. However, since a GUID is 16 bytes in size, it is impractical to use the GUID as the tag since it would make the ISF stream extremely large. Instead, ISF uses an index value as the tag that actually maps into a GUID lookup table.

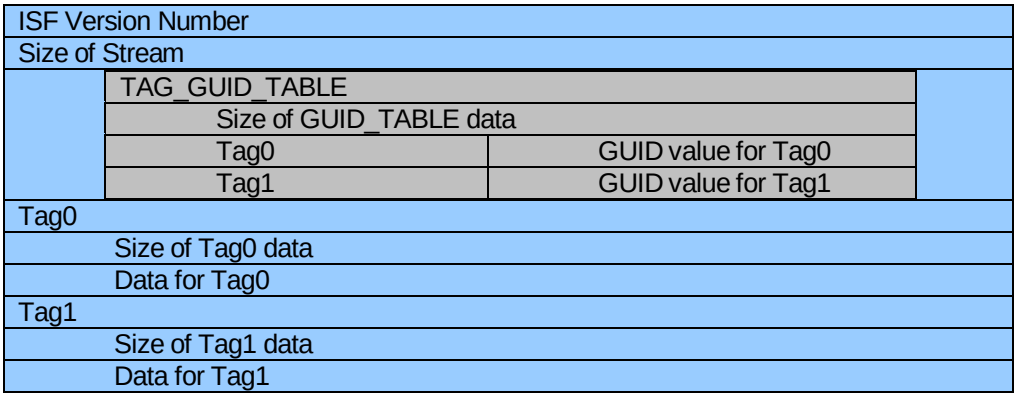

Figure **Error! Bookmark not defined.** - ISF with GUID table

This lookup table is then stored in the serialized stream as just another global ink property. This means that the actual tags in the serialized stream can be a small integer index and very little space is wasted on the GUID since it is stored only once in the stream.

The first 100 index values are reserved by ISF and are assigned predefined GUID values. These GUID and index values are listed in the appendix under predefined tags. These GUID values represent generally useful ink properties such as color, pressure, etc. Since these properties are predefined, their corresponding GUIDS are not actually stored in the lookup table. Instead an ISF implementation knows that when it sees one of these predefined tags that it corresponds to a predefined GUID. Instead only custom application-defined GUID values are stored in the lookup table in the ISF. In fact, there will never be a GUID table in the stream unless the application actually needs custom properties.

ISF makes similar optimizations by taking resources that may apply to multiple strokes, such as a drawing attribute, and only storing it once in the ink stream using a similar table strategy. For example, a drawing attribute, which describes how a stroke is to be rendered, is stored in a table. This table is then stored as a global ink property. This means that only an index is stored with each stroke instead of the whole drawing attribute. A similar approach is taken for other shared resources that may apply to multiple strokes. Again these additional tables and properties will be described in detail later.

SIMPLE EXAMPLE

Now let us illustrate these basic concepts by laying out a sample stream for a simple ink object which contains only x and y data in each strokes. An ISF requirement was that these simple streams be stored very efficiently. The ink stream below contains two strokes with no additional properties:

0	WISP ISF 1.0 version number
cbInkObject	Size of the stream in bytes from TAG_STROKE onward.
TAG_STROKE	Tag for the stroke
cbStroke	Size for stroke in bytes from cPoints onward
cPoints	Count of points in this stroke
X data	X coordinates (maybe compressed)
Y data	Y coordinates (maybe compressed)
TAG_STROKE	Tag for stroke object
cbStroke	Size for stroke in bytes from cPoints onward
cPoints	Count of Points in this stroke
X data	X coordinates (maybe compressed)
Y data	Y coordinates (maybe compressed)

Figure **Error! Bookmark not defined.** - ISF Simple Example

Both strokes in the above example are of the simplest form since they contain only the count of points (cPoints), and the compressed x and y data for each stroke. This is defined as the *default stroke structure*. That is, there are no additional packet properties (such as pressure) associated with each point, nor any other per stroke information in any of the strokes. A conforming ISF implementation would render this ink with the default drawing attributes (black pen, width of one, ball tip, etc.)

The ordering of the values for cbStroke, cPoints, X and Y data are always stored in the above order for a default stroke.

Note that the count of bytes used to store the compressed x data or compressed y data is not stored in the ISF. This is because given cPoints, the decompression algorithm can just read enough bytes from the stream until it decompresses cPoints number of coordinates. The first byte of the X and Y data indicates the compression algorithm used, if any.

COMMON STREAM ITEMS

Before we examine a more complex ISF stream, we must first introduce the common tags, and concepts that are found in a more complex ISF stream. The following section introduces the tags and concepts that are necessary to understand a more complex example.

The Version

The very first item in an ISF stream is the ISF version number. For version 1.0, the value will be zero.

Any conforming ISF 1.0 implementation will immediately stop processing a stream where this version number is not zero, create an empty ink object and return an error to the application.

Size of Stream

Immediately following the version number is the size field for the stream. The value of this field indicates the total number of bytes following the size field.

A conforming implementation must be prepared to accept a size value equal to the largest file size on Windows NT. Today this is a 64-bit number but may be larger in the future. Obviously on some implementations, such as ISF on a Windows CE device, it would not be reasonable to expect an application to be able to process a file of this size. The implementation in that case would generate an appropriate error to the application.

A side benefit of having the size of the stream actually stored in the stream is that if an application passes a random buffer to an ISF implementation, then the first two values in the file will usually be zero. This means that an ISF implementation will generally immediately stop processing an invalid stream.

Global Ink Properties

The section immediately following the size field contains the Global Ink Properties. For simple streams, there may be no Global Ink Properties. However, if there are Global Ink Properties, then they must appear before any Strokes and apply to the entire stream. This section of the document describes the common global properties. These properties will usually appear in the order they are presented below if they appear in the stream, with one exception.

GUID Table

The GUID table, if it appears in the stream, must appear immediately after the size field.

The GUID table is identified with the tag TAG_GUID_TABLE. The size of this table is a multiple of 16 bytes, where each entry is a custom GUID as used in the stream.

Ink Space Rectangle

The tag TAG_INK_SPACE_RECT identifies the Ink Space Rectangle, when present in the stream. It does not have a size field since it has a fixed size of four signed numbers.

These four numbers represent the left, top, right, and bottom of the ink space. The Ink Space Rectangle defines the virtual coordinate space for the ink. An application uses this rectangle to determine what area of the ink to either display or print. The Ink Space Rectangle essentially defines a virtual sheet of paper that the ink is drawn on. This does not mean that ink may not appear outside this area. However, the application will use this rectangle when deciding how to display the ink in a Window or on the printer.

Drawing Attributes Table

The Drawing Attributes table was briefly mentioned in an earlier section. This table lists all Drawing Attribute Blocks in the stream. Each Drawing Attribute Block defines information used while rendering the ink. These blocks may apply to one or more strokes and are placed in this table so that they are not repeated in each stroke.

The tag, TAG_DRAW_ATTRS_TABLE, identifies the Drawing Attributes table and is immediately followed by the size of the table. The size of the table is equal to the sum of the sizes of all Drawing Attribute Blocks.

A Drawing Attributes Table containing only one Drawing Attributes block is a special case. The tag and size for the table is omitted and the entire table is replaced by a single Drawing Attributes Block.

Drawing Attributes Block

Each Drawing Attributes Block starts with the tag, TAG_DRAW_ATTRS_BLOCK, and is followed by the size of the block. The block contains a tagged list of drawing attributes. Each entry in the list of drawing attributes is a pre-defined or custom tag. The diagram below illustrates a drawing attributes block:

TAG_DRAW_ATTRS_BLOCK	Tag omitted if in Drawing attributes table.
cbDrawAttrsBlock	Size of this block
TAG_PEN_WIDTH	Predefined Pen Width Tag.
Pen width value	
TAG_COLORREF	Predefined Colorref tag.
COLORREF value	
TAG_CustomDrawingAttribute	Tag indication a custom Draw attribute.
cbCustomDrawingAttribute	Size of the custom drawing attribute data
Data	Compressed data for the custom drawing attribute

Figure **Error! Bookmark not defined.** - Drawing Attribute Block

To save space in the ISF, the tag for the block is omitted when the block appears in a Drawing Attributes Table. This is because the table can only contain blocks and the tag is redundant. The next block starts immediately after the scope of the previous one.

Furthermore, all predefined drawing attributes, such as Pen Width or Color Ref, have no size indicator since the data values are known ahead of time. However, a custom property must have a size field since the size of the data is not known ahead of time. In addition, a custom property's size field must be greater than zero. Note also that custom properties are also assumed to be compressed. The first byte of the data must indicate the compression type (see below). This byte is not included in the size that is stored in the property's size field.

Another space saving optimization is that drawing attributes that are set to the default value are not stored in the Attributes block. In the above example, pen tip value is not stored since it is assumed to be the default of PEN_TIP_BALL.

The custom drawing attributes are ignored by the WISP rendering routines. It is assumed that they will be used by custom ink rendering routines implemented in the application.

Stroke Descriptor Table

The Stroke Descriptor Table lists Stroke Descriptor Blocks in the stream. These blocks may apply to one or more strokes and are placed in this table so that they are not repeated in each stroke.

The tag, TAG_STROKE_DESC_TABLE, identifies the Stroke Descriptor Table and is immediately followed by the size of the table. The size of the table is equal to the sum of the sizes of all Stroke Descriptor Blocks.

A Stroke Descriptor Table containing only one Stroke Descriptor Block is a special case. The tag and size for the table is omitted and a single Stroke Descriptor Block replaces the entire table.

Stroke Descriptor Block

As was illustrated in our simple example earlier, a stroke contains arrays of data where each array element corresponds to a property of a point. Our simple example contained only X and Y data since there was no stroke descriptor block in the stream. However, an application may wish to store other properties, such as pressure. An application could simply create a custom stroke property (described later) to store pressure. Unfortunately no other application would know how to interpret this data and the tag and size of the custom property would be stored in each stroke, thus wasting space in the ISF.

The purpose of the Stroke Descriptor Block is to solve this problem defining the data types and their order in the stroke. ISF can then use an index to associate a stroke with a particular Stroke Descriptor Block. This index is described later.

Typically all strokes in an ISF stream will use the same Stroke Descriptor Block, which is why a Stroke Descriptor Table contains only one block as a special case. However, ISF allows different strokes to contain different sets of data by placing the blocks in a table.

Each Stroke Descriptor Block starts with the tag, TAG_STROKE_DESC_BLOCK, and is followed by the size of the block.

The diagram below illustrates a Stroke Descriptor Block:

TAG_STROKE_DESC_BLOCK	Tag omitted when block is in the Stroke Descriptor Table
cbStrokeDescriptorBlock	Size of the whole stroke descriptor block.
TAG_NO_X	Optional, if this tag is present stroke contains no x data
TAG_NO_Y	Optional, if this tag is present stroke contains no y data
Array of packet property tags	Each entry defines a property tag
TAG_BUTTONS	Button descriptions in the packet, if any
cButtons	cButtons is the count of button states
Array of cButtons button tags	Button GUID tags
TAG_STROKE_PROPERTY_LIST	After this tag, the tags for global stroke properties are listed until the end of Stroke Descriptor Block.
Array of tags for stroke properties	Continues until run out of cbStrokeDescriptorBlock

Figure **Error! Bookmark not defined.** - Stroke Descriptor Block

It is assumed that, by default, all strokes will contain X and Y coordinate data arrays and that these will be the first data arrays stored in a stroke. If for some reason the application does not wish to store X and Y coordinates, then it needs to create a Stroke Descriptor Block containing the TAG_NO_X and TAG_NO_Y values as the first two entries. Otherwise the first two arrays in a stroke are always assumed to correspond to the X and Y coordinates.

Immediately after the optional TAG_NO_X, TAG_NO_Y is an array of packet property tags. This array ends when TAG_BUTTONS, TAG_STROKE_PROPERTY_LIST or the end of scope is encountered. Each packet property tag defines another array in the stroke.

Following the Packet Property Array is an optional Buttons Section that describes the button bit-fields that make up the elements of the button array in the stroke. Not all input devices report button states, so this section is optional. If present, the Buttons Section starts with TAG_BUTTONS followed by the count of buttons (cButtons) and an array of button GUID tags, one tag for each button. Note that these tags may be encoded (described later) and the size of the button array may not be an exact multiple of the number of buttons (cButtons).

If the end of the Stroke Descriptor Block scope has not been reached, then what follows is a TAG_STROKE_PROPERTY_LIST followed by a listed of stroke property tags. These tags do not describe arrays of values. Instead they are an optimization that allows a tag that appears repeatedly in strokes to be omitted; only the size and the data need to be specified for the listed property. A stroke may still have additional stroke properties that are not listed in its Stroke Descriptor Block under TAG_STROKE_PROPERTY_LIST. However, these additional properties have to be listed in the stroke *after* all the properties listed in the block and they do must be tagged explicitly within the stroke.

Transform Table

Several problems occur when ink is transformed. The first problem is the assumptions made when designing the compression schemes may no longer be valid. This may result in data that would have been compressed well in its native form to bloat and compress very poorly. The second problem is that it is very easy to perform operations on the ink

that would cause precision to be lost. This may cause the recognition process to fail and prevent the ink from being converted to text or might cause the ink to render incorrectly.

To solve this problem, ISF allows the ink to be stored in its original native format. Whenever ink is transformed, only a transform matrix is affected rather than each point in the ink. This preserves the precision of the original ink and also allows compression to function optimally.

The Transform Table lists all transform Blocks in the stream. Each Transform Block defines a unique transform that must be applied to the ink points before they are rendered or used. These blocks may apply to one or more strokes and are placed in this table so that they are not repeated in each stroke.

The tag, TAG_TRANSFORM_TABLE, identifies the Transform table and is immediately followed by the size of the table. The size of the table is equal to the sum of the sizes of all Transform Blocks.

A Transform Table containing only one Transform block is a special case. The tag and size for the table is omitted and a single Transform Block replaces the entire table.

In the simplest case and the cases where scaling and transforms have been applied outside of WISP (MSWord is an example) there will be no transform table since the transforms have already been applied to all points. In these cases compression may suffer.

Transform Block

The Transform Block is different than the Stroke Descriptor Block and Drawing Attributes block since there are several different transforms and thus several different tags.

The generic transform is defined by the TAG_TRANSFORM and is followed by 6 floats, corresponding to the matrix values m11,m12,m21,m22, dx and dy.

The other transform tags are:

- TAG_TRANSFORM_ISOTROPIC_SCALE
  M11 = M22 nonzero value; M12 = M21 = DX = DY = 0;
- TAG_TRANSFORM_ANISOTROPIC_SCALE
  M11, M22 arbitrary non zero floats; M12 = M21 = DX = DY = 0;
- TAG_TRANSFORM_ROTATE
  Storing MBE integer number, 0 – 36000 // 1/100 of degree units -
- TAG_TRANSFORM_TRANSLATE
  M11=M22=M12=M21=0; DX, DY arbitrary
- TAG_TRANFROM_SCALE_AND_TRANSLATE
  M11, M22, DX,DY arbitrary; M12=0,M21=0

Metric Table

The Metric Table lists Metric Blocks in the stream. These blocks may apply to one or more strokes and are placed in this table so that they are not repeated in each stroke.

The tag, TAG_METRIC_TABLE, identifies the Metric Table and is immediately followed by the size in bytes of the whole table, which consists of Metric Blocks. The size is then followed by the Metric Blocks. When the Metric Blocks are stored in the Metric Table, the tag TAG_METRIC_BLOCK that identifies them is omitted.

A Metric Table containing only one Metric Block is a special case. In this case the tag and the size of the table is omitted and a single Metric Block replaces the entire table. In that case the metric block needs to be tagged with its own tag, TAG_METRIC_BLOCK.

Metric Block

Our earlier example showed how an ISF stream may contain only strokes made up of (X,Y) point data. The Stroke Descriptor then enabled the addition of more properties to a stroke. The Metric Block further refines the definition of the properties defined in a stroke. The individual array elements in a stroke, such as the X or Y values, are represented in logical device coordinates. However there will be times where an application will need to know the relationship between these logical values and some real physical characteristics. For example is pressure in pounds, Pascal's or kilograms, or is an angle with a value of 10 in degrees or radians? Without further information an application must assume these values are in the standard normalized form, as defined by the pen system.

The purpose of the Metric Block is to define the relationship between the logical units stored in the stroke and physical characteristics. The most common ones being: Logical Minimum value, Logical Maximum value, Resolution, and Units.

Typically all strokes in an ISF stream will use the same Metrics Block. An ISF stream may even have several Stroke Descriptors, yet still only may have one Metric Block. However, ISF allows different strokes to refer to different Metric Blocks in the Metric Table.

The Metric Block itself is a table that consists of Metric Block Entries. Each Metric Block Entry corresponds to a packet property for which the metrics needs to be defined and at least one values is different from the default values. Therefore the Metric Block has the following form:

TAG_METRIC_BLOCK	Omitted if in table.
cbMetricBlockSize	Total size in bytes of all entries
Entry[0]	Data for the first Metric Block Entry
....	More entries
Entry[cMetricBlockEntries - 1]	Data for the last Metric Block Entry

The diagram below illustrates a single Metric Block Entry:

TAG_packet_property	Tag of the packet property this entry is describing
cbSize	The size of the remainder of the structure
Logical Min Value	Signed multi-byte encoded.
Logical Max Value	Signed multi-byte encoded.
Units Value	Byte value. See Spec Shared Definitions for the set of allowed values.
Resolution Value	FLOAT value, not encoded
More Data	Perhaps more data, depending on cbSize

Figure **Error! Bookmark not defined.** - Metric Block Entry

The Metric Block does not need to be defined for all the packet properties in any given stroke since the application may not care about the metrics associated with all the properties or the device may not provide metrics for all the properties. For example, it is not useful specifying minimal and maximal values for GUID_PACKET_STATUS as this is a flag field.

The Metric Block differs from the Stroke Descriptor since it may contain a TAG_X and a TAG_Y. This is because X and Y values may have metrics that need to be stored.

The table below defines default Metric Blocks for predefined packet properties.

Packet Property Tag	Min	Max	Unit	Resolution
TAG_X	0	12699	CENTIMETERS	1000
TAG_Y	0	9649	CENTIMETERS	1000
TAG_Z	-1023	1023	CENTIMETERS	1000
TAG_NORMAL_PRESSURE	0	1023	DEFAULT	1
TAG_TANGENT_PRESSURE	0	1023	DEFAULT	1
TAG_BUTTON_PRESSURE	0	1023	DEFAULT	1
TAG_X_TILT_ORIENTATION	0	3600	DEGREES	10
TAG_Y_TILT_ORIENTATION	0	3600	DEGREES	10
TAG_AZIMUTH_ORIENTATION	0	3600	DEGREES	10
TAG_ALTITUDE_ORIENTATION	-900	900	DEGREES	10
TAG_TWIST_ORIENTATION	0	3600	DEGREES	10
TAG_PACKET_STATUS	NA*	NA	NA	NA
TAG_TIMER_TICK	NA	NA	NA	NA
TAG_SERIAL_NUMBER	NA	NA	NA	NA
TAG_PITCH_ROTATION;	ND**	ND	ND	ND
TAG_ROLL_ROTATION;	ND	ND	ND	ND

TAG_YAW_ROTATION;	ND	ND	ND	ND

Figure – Table of default Packet Description values for predefined packet properties.

*NA: Means that metric values do not make sense or that they are irrelevant and do not need to be stored in the serialized format.

**ND: Metric Values are important but WISP does not define defaults, the values need to be queried from the tablet and stored in the serialized format.

If a Metric Block Entry for a given packet property is missing it is assumed that the metrics values for that property are default values or if the default values are not defined, then metric values are not important. Similarly, if a metric block for a given property is present, but it only contains fields up to and including say max value, but not units and resolution, then it is assumed that units and resolution are the same as in the corresponding default block, but min and max may be different. For example, the values for TAG_X, TAG_Y in the default table correspond to a WACOM Intuous tablet with approximate dimensions 12.70 cm x 9.65 cm. Another WACOM tablet with different dimensions is likely to have the same min value of zero, units and resolution, but perhaps a different max values of x and y. Therefore the Metric Block Entry for x for such a tablet may look something like this:

TAG_X	Tag of the packet property this entry is describing
cbSize	Only includes min and max values.
0	Min value of X, encoded in one byte
Max Value different from the default	Signed multi-byte encoded.

Bit Assignment Table

Bit assignment table starts with the multi-byte encoded count of Bit Assignment Blocks. Bit Assignment Blocks follow the count. If the count of BABs is zero, no Bit Assignment Blocks follow, but the count still needs to be stored. However, in this case Index Map Table has at least one Index Map Block.

Bit Assignment Block

Bit Assignment Block (BAB) is a monotonically increasing array of numbers in the range [0,32] where the first number is always 0 and the last is 32. The first element 0 and the last element 32 are not stored explicitly in the array. This array specifies the Huffman prefixes, that is, how many bits are used to represent absolute delta-delta values in the compressed array.

WISP defines a set of eight default BABs, which are hard coded in the compression module. When additional BABs are needed, they are stored as bit-packed array. Given that no member of this array is bigger than 32, 5 bits suffice to store each member. In general, for a BAB that has cEntries, (cEntries * 5 + 7)/8 bytes are needed to store the whole array in a bit-packed format. Graphically, this can be represented as follows:

Data	Comment
cEntries	Count of entries in the BAB array, not counting 0 and 32. Fits in a single byte.
BAB array	(cEntries * 5 + 7)/8 bytes of bit-packed data

Index Map Table

Index Map Table starts with the count cIMB, count of Index Map Blocks, followed by that many Index Map Blocks.

Index Map Block

Index Map Block describes the Index Mapping, the mapping that describes the ordering of the absolute packet data delta-delta values in the order of monotonically decreasing probabilities. When absolute delta-delta values are reordered in such a way, it turns out that the distribution of delta-deltas conforms a standard zero-mean normal distribution to within a very small statistical error. (Gaussian distribution centered at zero).

In practice, if the stroke data points in the ink object are collected directly off the collection device, without performing any subsequent scaling of the x,y data, the Index Mapping is trivial and reordering is not necessary. When x or y data has been scaled out by a factor **s** before storing it in the ink object, the distribution of absolute delta-delta values will have narrow but finite width spikes centered around n*__s__, where n is an integer. In this case Index Mapping is needed to reorder delta-delta values in order of monotonically decreasing probabilities.

Index Mapping is a variable length array of (delta-delta) packet data. We store the Index Mapping in an Index Map Block in the following format:

Data	Comment
iBAB	One byte to identifies the BAB to be used with this Index Map. If this number is less than 8 (the number of default BABs available), then it indicates the index of the default BAB. Otherwise, this number minus 8 is the index of the custom BAB defined in the Bit Assignment Table.
cCount	MBE integer indicating number of elements in this Index Mapping
Index Map Array	MBE integer elements of this Index Mapping. Each element is an absolute delta-delta value. The elements are stored in order of decreasing probabilities.

Custom Global Ink Properties

Custom Global Ink properties are application-defined properties that apply to the entire ink stream. It is up to the application to interpret what these custom properties mean. ISF simply stores these properties one after the other. The following is an example of the custom global ink property embedded in the stream:

…..	Preceding data in the stream
TAG_FOO	Tag indexing the Custom GUID in the GUID table
cbFoo	Count of bytes for the compressed data for property Foo. The size does not include algorithm byte.
Data for Foo	Compressed data for Foo
…..	More data in stream

Figure **Error! Bookmark not defined.** - Custom Global Ink Property

Local Properties

Local properties come after the global ink properties. Local properties do not apply to the entire ink stream. Strokes are an example of a local property. Other local properties, such as a Drawing Attribute Index apply to all the strokes that appear after that point in the stream until the next time that local property appears in the stream again. Just like Global Ink Properties, local properties are also optional. A valid ISF stream could potentially contain no local properties at all. However, such a stream would not be too useful.

Drawing Attribute Index

The Drawing attribute Index (TAG_DIDX) assigns a Drawing Attribute Block to a stroke. TAG_DIDX is followed by an index value that specifies the entry in the Drawing Attributes Table. All strokes in the stream from that point on use the specified Drawing Attribute Block until the next TAG_DIDX is encountered in the stream.

As an additional optimization, if there is no TAG_DIDX in the stream somewhere before the stroke, it is assumed that this stroke should use the 0^{th} Drawing Attribute Block in the Drawing Attributes Table. And if there is no Drawing Attributes Table in the stream, then all strokes should be drawn using the default set of drawing attributes.

Stroke Descriptor Index

The Stroke Descriptor Index (TAG_SIDX) assigns a Stroke Descriptor Block to a stroke. TAG_SIDX is followed by an index value that specifies the entry in the Stroke Descriptor Table. All strokes in the stream from that point on use the specified Stroke Descriptor Block until the next TAG_SIDX is encountered in the stream.

As an additional optimization, if there is no TAG_SIDX in the stream somewhere before the stroke, it is assumed that this stroke should use the 0^{th} Stroke Descriptor Block in the Stroke Descriptor Table. And if there is no Stroke Descriptor Table in the stream, then all strokes are assumed to contain X and Y coordinates only.

Transform Index

The Transform Index (TAG_TIDX) assigns a Transform Block to a stroke. TAG_TIDX is followed by an index value that specifies the entry in the Transform Table. All strokes in the stream from that point on use the specified Transform Block until the next TAG_TIDX is encountered in the stream.

As an additional optimization, if there is no TAG_TIDX in the stream somewhere before the stroke, it is assumed that this stroke should use the 0^{th} Transform Block in the Transform Table. And if there is no Transform Table in the stream, then no transforms are applied to any stroke.

Metric Index

The Metrics Index (TAG_MIDX) assigns a Metrics Block to a stroke. TAG_MIDX is followed by an index value that specifies the entry in the Metrics Table. All strokes in the stream from that point on use the specified Metrics Block until the next TAG_MIDX is encountered in the stream.

As an additional optimization, if there is no TAG_MIDX in the stream somewhere before the stroke, it is assumed that this stroke should use the 0th Metrics Block in the Metrics Table. If there is only one Metrics Block then the table is omitted and the 0th Metrics Block is the only Metrics block in the stream.

Strokes

As described earlier in the simple example, Strokes are the most fundamental and important property in ISF. Strokes contain the packet data that make up the individual points in a stroke and potentially other per-stroke properties as well.

The diagram below illustrates a typical stroke.

TAG_STROKE	Tag for the stroke
cbStroke	Size for stroke in bytes (including cPoints and onward)
cPoints	Count of points in this stroke
X data	X coordinates (maybe compressed)
Y data	Y coordinates (maybe compressed)
Buttons	Array of button state bitmaps.

Figure **Error! Bookmark not defined.** - Simple Stroke Example

The stroke begins with a TAG_STROKE and is immediately followed by a size field that is the count in bytes (cbStroke) of all the data starting from (and including) cPoints to the end of the stroke. cPoints immediately follows cbStroke and defines how many points are stored in the packet data of this stroke.

Packet Data

The packet data starts immediately after cPoints. The packet data consists of arrays of values, where each array is in the same order as described in the stroke descriptor for the stroke. If the stroke descriptor did not specify TAG_NO_X or TAG_NO_Y, then the first two arrays are the X and Y coordinate arrays. It is possible to define a stroke descriptor that contains no packet data at all and hence no points. In that case cPoints will be zero.

These packet data arrays are followed by an array of button state bitmaps. Each button state for each point is stored as a single bit in this array. The button states are stored in a bit-packed array. The number of buttons cButtons is read from the stroke descriptor. The buttons array will require ((cPoints * cButtons+7)/8) bytes in the stream to store all the state information for each button and for all points.

Note that buttons states in the non-serialized ink packets require ((cButtons + 31)/8) bytes for each packet, i.e. DWORD rounding is used. However, when button states for the whole stroke are serialized, storage-wise more efficient bit-packing is used, where the whole bit-array is rounded up once to a byte boundary.

Stroke Properties

Following the packet data are Stroke properties, if any. Stroke properties that are listed by the TAG_STROKE_PROPERTY_LIST appear immediately after the buttons array and have their tags omitted since they are specified in the stroke descriptor.

If the end of the stroke scope has still not been reach then more stroke properties may be specified listing them with their tags.

Point Property List

The point property list is just a stroke property. A Point Property List starts with the TAG_POINT_PROPERTY and the size of the whole Point Property List. A Point Property List describes properties that are attached to specific points. The table below illustrates a typical point property list.

TAG_POINT_PROPERTY	Tag of the Point Property List
cbPointProperty	Size of the whole Point Property List
tagFoo1	Property foo1
Index1	Index of the point within the stroke
cbFoo1	Size of data for property foo1
Data for Foo1	Data for foo1, compressed
tagFoo2	Property foo2
Index2	Index of the point within the stroke
cbFoo2	Size of data for property foo2
Data for Foo2	Data for foo2, compressed

Figure **Error! Bookmark not defined.** - Point Property Example

The Point Property List starts with the size of the whole list and is followed by the list of tagged point properties. Each tag has an index that refers to a point in the stroke this tag applies to, followed by the size of the data (without algorithm byte) and the data itself.

This above example list has two properties. The property Foo1 is “attached” to the point identified by Index 1 while the property Foo2 is “attached” to the point identified by Index2.

COMPLEX EXAMPLE

This section now illustrates a more complex ISF example using some of the tags and concepts from the preceding section. The example below describes an ink object with three strokes where one of the strokes contains pressure values and the others do not:

0	ISFVersion number, set to zero
cbInkObject	Size of the whole object
TAG_STROKE_DESC_TABLE	Tag for stroke descriptor table
cbStrdTable	Size of stroke descriptor table containing two blocks
cbStrdBlock0 = 0	Size of the 0th block is 0 length, only x,y in these strokes.
cbStrdBlock1 != 0	Size of the 1st stroke descriptor block
TAG_NORMAL_PRESSURE	Indicates pressure is the first thing after x,y
TAG_METRICS_BLOCK	The only metrics block in the stream.
cbMetricsBlock0	Size of Metrics block.
TAG_NORMAL_PRESSURE	Metrics for X and Y are not stored in this example.
cbMetricValues	
TAG_MIN	Minimal allowed value for pressure, eg -100
10	No value in the stream will be below 10
Max Pressure	Maximal allowed value for pressure eg +100
90	No value in the stream will be above 90
Precision	Precision of device. eg. 100 per unit.
1	Precision of device logical units are in increments of 1
Units	Units for the precision. eg value indicating Tons.
2	Numeric value indicating units. If this number indicated that the units where in kilograms then because precision is 1 then each increment of logical units would be a kilo.
TAG_SIDX	Tag for index into a stroke descriptor table
1	Value for the stroke descriptor index
TAG_STROKE	Tag for the stroke
cbStroke	Size for stroke and all its children
cPoints	Count of points in this stroke
X data	Compressed X coordinates
Y data	Compressed Y coordinates
Pressure data	Compressed normal pressure data
TAG_SIDX	Tag for index into a stroke descriptor table
0	Value for the stroke descriptor index
TAG_STROKE	Tag for stroke object
cbStroke	Size for stroke and all its children
cPoints	Count of Points in this stroke
X data	Compressed X coordinates
Y data	Compressed Y coordinates
TAG_STROKE	Tag for stroke object
cbStroke	Size for stroke and all its children
cPoints	Count of Points in this stroke
X data	Compressed X coordinates
Y data	Compressed Y coordinates

Figure **Error! Bookmark not defined.** - Complex ISF Example

At the beginning of the ISF stream, before any strokes, there is a Stroke Descriptor Table with two Stroke Descriptor Block entries. The 0th entry in the Stroke Descriptor Table has size zero, i.e. it is empty. This signifies that the strokes which are described by this entry contain only x and y vectors. The 1st entry of the stroke descriptor table describes the strokes that in addition to x and y arrays contain a pressure vector. Note that this entry does not contain descriptors for x and y arrays, they are implicit, only pressure is described with its minimal and maximal values.

Next comes the tag TAG_SIDX followed by the index value. The Stroke Descriptor Table Index value refers to all the strokes in the ink object that follow in the stream, until the end of the ink stream or until a next Stroke Descriptor Table Index is found in the stream.

In the example above Stroke Descriptor Table Index value of 1 applies to the 0th stroke and the Stroke Descriptor Table Index value of 0 applies to the 1st and 2nd strokes. Therefore, the 0th stroke contains pressure vector in addition to x and y vectors, the 1st and 2nd strokes only contain x and y vectors.

ENCODING

Another one of the ISF requirements was that it must be efficient to store. Tags are indexes to GUIDs so that the GUID is not repeated unnecessarily. The previous section mentioned that the X and Y data might be compressed. In fact, these are all compression strategies. To achieve the goal of efficient storage, a number of encoding strategies and compression methods are used.

Sizes of Tags and Numbers

At the most basic level, ISF is composed of numbers. Even tags are indexes, which are just small integer numbers. In fact, most of the time these numbers are small enough that they could be represented by a single byte if there was a way of determining when a byte represented a single number and when it was just part of a bigger number. It is possible to take advantage of this observation by encoding numbers using a multi-byte encoding technique.

Multi-byte encoding makes it possible to represent small numbers in one byte, larger numbers in two bytes and very large numbers in however many bytes are necessary.

This means that tags, which usually have a value less than 100, are stored as a single byte and sizes, which may be small or large, are stored in the most efficient manner. In effect, multi-byte encoding is a compression technique.

Multi-byte encoding works as follows:

- Numbers less than 128 are encoded in one byte.
- This leaves the most significant bit in the byte clear.
- Multi-byte encoding interprets the most significant bit being clear to mean this is the last byte in a number.
- Numbers larger than 128 are broken up into 7 bit segments.
- These 7 bit segments are then each stored in a byte.
- And the most significant bit in each byte except the last is set.

This means that

- Numbers less than 2^7 = 128 are encoded in a single byte.
- Numbers less than 2^14 = 16384 are encoded in two bytes.
- Numbers less than 2^21 = 2097152 are encoded in three bytes.
- Etc.

In general, bytes are processed until a byte with the most significant bit clear is encountered.

For example, the first number encountered in ISF is the version number. For version 1.0 this value is “0” and can be encoded in a single byte. Next number is the size of the stream following the size value, and for small ink objects as in the first example this will also be encoded in a single byte. However, if the stream is long this value can grow as large as necessary. For example a multi-byte encoded number of 10 bytes can represent a 64-bit number.

This same process is applied to “tags”, and other values in the stream. In general since “tags” are small integer indexes, they too will be one byte encoded in most cases.

Multi-byte Encoding of Signed Numbers

Multi-byte encoding as described above works well for positive integers however in some cases it is necessary to store signed numbers. For example the coordinates of a point may be positive or negative depending on where the application situates the origin.

To multi-byte encode a signed number, the absolute value of the signed number is determined, the absolute value then is shifted left by 1 bit, and the sign of the original number is stored in the least significant bit.

Using this technique, the signed numbers with absolute values:

- less than 2^6 = 64 are encoded in one byte,
- less than 2^13 = 8192 are encoded in 2 bytes
- etc.

Tags for Predefined and Custom GUIDs

Representing tags with indices into a GUID table is another technique used to make the ISF efficient. As was already mentioned above, some of these GUIDS are predefined by ISF while others are custom to the application. The first 100 entries in the table are reserved by ISF and are not stored in the serialized stream.

Only custom GUIDs, which are represented by indices greater than 100, are stored in the serialized format. The tag TAG_GUID_TABLE in the stream identifies the Custom GUID look up table. The custom GUID table is part of the ink stream only if the ink object contains at least one custom GUID.

As mentioned in the previous section, ISF uses multi-byte encoding for these tags. As long as an application does not use more than 27 custom properties in any given ink object, then the all tags will be encoded using only a single byte.

Compression

ISF provides a number of compression algorithms to take advantage of specific properties of certain types of data that appear in the stream. Generally, the ISF algorithms belong to two classes, packet data compression and general property data compression.

Packet Data Compression

One class of algorithms is used for packet arrays, such as x, y, pressure etc.

The X and Y coordinates, which make up the bulk of the ink stream, have the interesting property that one coordinate is usually located close to the previous coordinate. In fact the difference (first derivative) between any two consecutive coordinates is usually a very small number, and the difference between the differences (second derivative) is usually an even smaller number. While this is not true of all strokes, it is true for most of the ink that is stored with ISF. For example, a stroke that is a straight line has a constant first derivative while the second derivative is zero. ISF uses this observation to store coordinates compactly. The coordinates are pre-processed and the first and second derivatives are determined. An array is then built where the first element is the first coordinate, the second element is the difference between the first two coordinates, and the remaining elements are the second derivative. ISF then uses an improved and finely tuned version of the Huffman algorithm to compress this array. The Huffman algorithm uses a table that is specifically tuned to work best when the data contains many small numbers. In our straight-line example, most of the array will have the value zero. Even with more complex data, the second derivative values will generally be zero or one. The compressed representation of this array will be extremely compact for data of this form. ISF then stored this array since the original data can be recalculated from this information. This method is greatly improved yet similar to the method used by PenWindows.

Property Data Compression

The second class of compression algorithms is applied to custom ink property data, or stroke property data or a custom drawing attribute. Basically, the data is an amorphous array of bytes whose structure is unknown to ISF.

One of the algorithms that can be applied in this situation is the LZ compression algorithm. This is the same algorithm and code that is used in other parts of the Windows operating system. LZ should provide a reasonable amount of compression on arbitrary data provided by the application.

For relatively short arrays, simple bit packing techniques may be faster and more efficient. These algorithms are discussed in more detail below.

Unfortunately, there will be times when none of the compression algorithms will compress the data. For example, if the application has pre-compressed a custom property before it passes to WISP for serialization, then no compression algorithm will be likely to further compress the data. In those cases, the data will be stored in its original form and marked as uncompressed. Since the first byte of the data is the compression algorithm indicator, the serialized data will actually be one byte larger than the original data.

It is critical that the ISF 1.0 compression algorithms work well on most ink data. This is because it is impossible to add new compression algorithms to ISF without creating an incompatibility. Adding or changing compression is one of the few reasons for

incrementing the ISF version number since a 1.0 implementation would not be able to read the stream compressed with a new algorithm. (If the compressed data was not critical or was a custom application property then the 1.0 implementation could ignore that data without serious loss of functionality.)

.

As already mentioned, any data that may be compressed begins with a one-byte compression algorithm identifier. We are now about to discuss in more detail the compression algorithms used by WISP and their one-byte identifiers.

Compression Algorithms for Property Data

Each compressed buffer is prefixed by one byte to identify the compression algorithm used in the serialized data. Pictorially:

Algorithm (1 byte)	(Compressed) Data Bytes

In the identifier byte we want to store the algorithm type and algorithm specific data if applicable. Identifier bytes for property data algorithms used by WISP and the bit assignment for each algorithm types are as follows:

Algorithm	Bit assignment							
	7	6	5	4	3	2	1	0
PROPERTY_BIT_PACK_BYTE	0	0	0	D	D	D	D	D
PROPERTY_BIT_PACK_WORD	0	0	1	D	D	D	D	D
PROPERTY_BIT_PACK_LONG	0	1	D	D	D	D	D	D
LIMPEL_ZIV	1	0	1	0	X	X	X	X
DEFAULT_COMPRESSION	1	1	0	0	X	X	X	X
BEST_COMPRESSION	1	1	1	1	X	X	X	X

D indicates data bits required by the algorithm.

X indicates don't care bits. Compression module does not even look at these bits.

More detailed explanation follows.

PROPERTY_BIT_PACK_BYTE

In this scheme, we do bit packing of 8 bit unsigned byte values. The last 5 bits contain the index into the cBits-cPads Lookup Table (in the appendix) that determine two numbers (cBits, cPads) that are needed to decompress the compressed array. The meaning of these two numbers is as follows.

First we need to store how many bits are required to store the maximum absolute value found in the non-compressed data array. We call this number cBits. In case of BYTE array cBits can at most be 8 bits.

In order to decompress the data the decompression routine must know not only the total size of the compressed data, but also the number of items in the original array. The number of items in the original array can be easily inferred if we know the size of the compressed array, the number of bits cBits needed to store one item, **and the number of**

items cPads that could be stored in the unused bits of the last byte of the compressed array.

For example, let us say we have an array of 3 items, each requiring cBits = 2 bits. Non-compressed size is 3 bytes, where each byte contains 2 bits worth of information in the least significant two bits. Compressed size is one byte, where 6 bits contain the compressed information and 2 bits are unused remainder. One could use those two unused bits to store cPads = 1 entry. So by saying that in the two unused remainder bits one could store one entry, we are really saying that only 3 array entries are compressed in this byte.

There are 24 possible different combinations of (cBits, cPads) pairs that apply for bit packing of the BYTE array. There additional 8 entries in the cBits-cPads Lookup Table apply to bit packing of the WORD array and 16 more entries apply to bit packing of LONG array. Please refer to the appendix for more detail.

PROPERTY_BIT_PACK_WORD

In this scheme, we do bit packing of 16 bit unsigned values. The values (cBits, cPads) for this case are obtained from first 32 entries of the cBits-cPads Lookup Table. The index into this table is always smaller than 32 and therefore fits in 5 least significant bits of the algorithm byte.

PROPERTY_BIT_PACK_LONG

In this scheme, we do bit packing of 32 bit signed values. The values (cBits, cPads) for this case are obtained from all of 48 entries of the cBits-cPads Lookup Table. The index is always smaller than 48 and therefore fits in 6 least significant bits of the algorithm byte.

LEMPEL_ZIV

We don't need to store any algorithm specific information.

DEFAULT_COMPRESSION

When default compression is requested for property data for a given ink object, the ISF compression module will use one of the above 3 bit packing techniques that would produce the smallest compressed data. Lempel Ziv is not used in the default compression mode, as it is quite time consuming.

BEST_COMPRESSION

When best compression is requested for property data for a given ink object, the ISF compression module will usemulti-table Huffman compression..

Compression Algorithms for Packet Data

Algorithm	Bit assignment							
	7	6	5	4	3	2	1	0
PACKET_BIT_PACK	0	0	D	D	D	D	D	D
DEFAULT_COMPRESSION	1	0	D	D	D	D	D	D
BEST_COMPRESSION	1	1	1	1	X	X	X	X

PACKET_BIT_PACK

In this scheme, we do bit packing of signed values. We need to store how many bits are required to store the maximum absolute value found. This can at most be 32, thus we need 5 bits to represent this value. The count of items in the compressed array is stored once per stroke in the serialized format and is passed as an input to the decompression routine. Therefore, this number is not necessary to store with each compressed packet stream.

Note that storing 32 actually requires 6 bits, as its binary representation is 100000. However, bit packing a LONG array using 32 bits per entry really means no compression. Therefore we represent 32 by just storing 5 zeros in the data bits. That way, no compression can be treated as special case of PACKET_BIT_PACK algorithm, where the compression algorithm byte is set to zero.

Least significant 5 bits will indicate the number of bits required to store the maximum value. If the 6^{th} bit is set, (n-2) delta-delta's are compressed, otherwise, the raw data is compressed.

DEFAULT_COMPRESSION

For packet data, DEFAULT_COMPRESSION alwaysuses multi-table Huffman compression. .

BEST_COMPRESSION

BEST_COMPRESSION may use the default Huffman, or PACKET_BIT_PACK, depending on what works the best for each packet array.

Backus-Naur Form (BNF) Specification

The following table is a BNF representation of the WISP stream format. The idea behind using BNF notation is that there are many items in the file that have positional interdependencies and BNF is the best way to represent these relationships. File formats are often represented using BNF notation. For example, a STROKE never contains another STROKE. However if one simply represents the WISP stream format as just a set of tagged structures these semantics are missing and could lead to one implementation generating an invalid format that would not interoperate with another implementation.

For those not totally familiar with BNF, the notation is of the form:

<A>	::=	NULL
	|	"a" <A>
	|	"a" NumericData <A>

The <A> is a "Logical"- non-terminal item, which must be further interpreted. The above example should be read as:

- <A> can be NULL / nonexistent – this is different from a null string. If <A> is NULL then there is no <A>
- or the literal "a" - terminal

- or the literal “a”, followed by the terminal item ‘NumericData’, followed by still more <A>.

As can be seen these definitions are recursive and can be used when expressing list, order and relationships between items. The following quick examples demonstrate the above grammar:

- aaaa – is valid since <A> is “a” followed by more <A>
- a1a23a – is valid since <A> can be an “a” followed by a number, followed by another “a”, followed by yet another number followed by “a”s
- aba – is not valid since “b” is not valid in the grammar

NOTE: The following grammar is only meant to provide specific details. It is important to understand the entire document not just the BNF section.

<WISP INK SERIALIZED FORMAT>	::=	<VERSION> size <GLOBAL INK PROPERTIES> <LOCAL INK PROPERTIES>	The first item in the stream is always the ISF version. Followed by the size of the Global and Local Properties. The size is a 9 byte-encoded value. (64 bit size) The rest of the ISF can be logically separated into two halves. A set of global ink properties that refer to the entire ink object and local ink properties containing stroke data and possibly other property data. The Global Ink Properties must come before the body.
<VERSION>	::=	"0"	This specification applies only to the version 1.0 of ISF. Incrementing the version number means that the stream format has been changed in an incompatible fashion and all implementations that only understand this version must not read the stream. Instead they should create and empty INK object with the default settings and return an error.
<GLOBAL INK PROPERTIES>	::=	<GUID TABLE> <INK SPACE RECTANGLE> <DRAWING ATTRIBUTES TABLE> <STROKE DESCRIPTOR TABLE> <TRANSFORM TABLE> <METRICS TABLE> <MORE GLOBAL PROPERTIES>	In the simplest case there are no <GLOBAL INK PROPERTIES> in the stream. The <GUID TABLE> must appear immediately after the stream size field if it exists. If the stream contains no custom properties then there will be no <GUID TABLE>. In that case all tags and GUID indexes will refer to one of the predefined tags or GUIDS.
<MORE GLOBAL PROPERTIES>	::=	NULL	There may be no more custom properties.
	|	GUID_IDX**,** **[**size**],** DATA <MORE GLOBAL PROPERTIES>	Or the ink stream may contain custom global properties. The GUID_IDX is an encoded index that refers to a GUID in the <GUID TABLE> that identifies the custom property. The actually entry in the table is the value of (GUID_IDX – 100). The index is then followed by the encoded size of the data and finally the custom data itself. Note that [size] is omitted if GUID_IDX points to a predefined GUID that has a known fixed size data type.
<GUID TABLE>	::=	NULL	There could be no GUID table.
	|	TAG_GUID_TABLE, size <GUID LIST>	If the stream contains custom properties then the stream will also contain a <GUID TABLE>. This table will begin with a single byte with the value TAG_GUID_TABLE followed by the encoded size in bytes of the <GUID

			LIST>. The size will be a multiple of 16 since all GUIDS are 128 bits in length. The <GUID LIST> is not compressed.
<GUID LIST>	::=	GUID	The <GUID LIST> may contain only a single GUID or a number of GUIDS.
	|	GUID <GUID LIST>	
<INK SPACE RECTANGLE>	:=	NULL	There may be no INK Space Rectangle
	|	L, T, R, B	The <INK SPACE RECTANGLE> defines a rectangle that indicates to the application the preferred area of the ink that should be displayed in the drawing area on the screen or on the printer. The application may ignore this value and display some other area, however if this value is present then it should use this rectangle for determining the aspect ratio of the ink. If no <INK SPACE RECTANGLE> is specified in the stream then the aspect ratio of the ink is the same a standard VGA monitor and the application must decide what portion of the ink to display. The values are encoded. Note ink is not clipped to ink space, so it is possible to draw outside the space. Also note there is no size field since a rectangle is a known structure with a known size.
<DRAWING ATTRIBUTES TABLE>	::=	NULL	There may be no Drawing Attributes.
	|	TAG_DRAWING_ATTRIBUTE_BLOCK <DRAWING ATTRIBUTE BLOCK>	If there is only one drawing attribute then only the block is stored.
	|	TAG_DRAW_ATTRS_TABLE Size <DRAW ATTRIBUTE LIST>	The individual strokes in the ISF stream may refer to entries in the <DRAWING ATTRIBUTES TABLE>. These entries define how to actually draw the stroke. They define attributes such as color, width, etc. See the WISP documentation for a complete list of drawing attributes. The <DRAWING ATTRIBUTES TABLE> starts with the TAG_DRAW_ATTRS_TABLE tag. It is then followed by the encoded size field, in bytes, of the entire list of drawing attributes.
<DRAW ATTRIBUTE LIST>	::=	<DRAWING ATTRIBUTE BLOCK>	A <DRAW ATTRIBUTE LIST> must contain at least one entry.
	|	<DRAWING ATTRIBUTE BLOCK> <DRAW ATTRIBUTE LIST>	

<DRAWING ATTRIBUTE BLOCK>	::=	size <DRAWING PROPERTIES>	Encoded size of entire Draw Attribute.
<DRAWING PROPERTIES>	::=	GUID_IDX [size] value	GUID_IDX may point to a well-known or custom GUID. See documentation for list of predefined drawing GUIDS. These may be tags for properties such as color, width, etc or may be application specific properties.
	|	GUID_IDX [size] value < DRAW PROPERTIES >	[size] is omitted for pre-defined tags that have a known fixed size data type.
<STROKE DESCRIPTOR TABLE>	::=	NULL	May be no table.
		TAG_STROKE_DESC_BLOCK <STROKE DESCRIPTOR BLOCK>	If there is only one stroke descriptor then only the block is stored.
	|	TAG_STROKE_DESC_TABLE size <STROKE DESCRIPTOR LIST>	The <STROKE DESCRIPTOR TABLE> is a list of <STROKE DESCRIPTOR BLOCKS> that describe what data may appear in a stroke. The <STROKE DESCRIPTOR TABLE> starts with the TAG_STROKE_DESC_TABLE tag. It is then followed by the encoded size field, in bytes, of the entire list of stroke descriptor blocks.
<STROKE DESCRIPTOR LIST>	::=	<STROKE DESCRIPTOR BLOCK>	The list must contain at least one block.
	|	<STROKE DESCRIPTOR BLOCK> <STROKE DESCRIPTOR LIST>	
<STROKE DESCRIPTOR BLOCK>	::=	size <NO_X DESCRIPTOR> <NO_Y DESCRIPTOR> <PACKET DESCRIPTOR> <BUTTON DESCRIPTOR> <OTHER DESCRIPTORS>	A stroke descriptor block describes the data that appears immediately after the TAG_STROKE in the stroke itself. The size field is the encoded size in bytes of all the descriptors in the block. A size of "0" means the descriptor block indicates the stroke has only X and Y data. Note: there is no tag for a stroke descriptor block since the stroke descriptor table only contains stroke descriptor blocks and the tag would be redundant.
<NO_X DESCRIPTOR>	::=	NULL	NULL means X data is present in the stroke
	|	TAG_NO_X	TAG_NO_X indicates there is no X data in the stroke.
<NO_Y DESCRIPTOR>	::=	NULL	NULL means Y data is present in the stroke
	|	TAG_NO_Y	TAG_NO_Y Indicates there is no Y data in the stroke.

<PACKET DESCRIPTOR>	::=	NULL	TAG_NO_X, TAG_NO_Y followed by a NULL <PACKET DESCRIPTOR> means the stroke contains no per point data, only per stroke properties.
	|	<TAG_LIST>	Packet descriptor lists properties other than X and Y that appear in the per point data of the stroke (such as Pressure, angle, etc.)
<BUTTON DESCRIPTOR>	::=	NULL	NULL if the stroke contains no button information.
	::=	TAG_BUTTONS, ButtonCount <TAG LIST>	If button information is present in the stroke data then the TAG_BUTTONS is present, followed by the encoded size in bytes of the list of button GUIDS. Each button in the list will correspond to a bit in the data.
<TAG LIST>	::=	GUID_IDX	A GUID_IDX points to either a well known GUID or a custom GUID.
	|	GUID_IDX <TAG LIST>	
<OTHER DESCRIPTORS>	::=	NULL	There may be no other descriptors in the block.
	|	TAG_STROKE_PROPERTY_LIST <TAG_LIST>	Or there may be a list of descriptors that describe properties about the stroke. Note these indexes do not describe per point properties but rather per stroke properties. This is provided as a mechanism so that the application is not forced to repeat the GUID_IDX tag in each stroke for a custom property.
<METRICS TABLE>	::=	NULL	May be no table.
		TAG_METRICS_BLOCK <METRICS BLOCK>	If there is only one Metrics block then only the block is stored.
	|	TAG_METRICS_TABLE size <METRICS BLOCK LIST>	
<METRICS BLOCK LIST>	::=	<METRICS BLOCK>	The list must contain at least one block.
	|	<METRICS BLOCK> <METRICS BLOCK LIST>	
<METRICS BLOCK>	::=	Size	Size is sum of all Metrics Property entries.

		<METRICS PROPERTY ENTRY>	
<METRICS PROPERTY ENTRY>	::=	GUID_IDX Size <TAG_VALUE_PAIR LIST>	GUID_IDX indicates to which packet property these metrics apply, Size is the sum of all the Tag/Value pairs.
<TAG VALUE PAIR LIST>	::=	GUID_IDX value	A GUID_IDX points to either a well known GUID or a custom GUID. Signed multi-byte encoded value for this GUID_IDX.
	|	GUID_IDX value <TAG LIST>	
<TRANSFORM TABLE>	::=	NULL	
	|	< TRANSFORM BLOCK>	If there is only one Transform then only the block is stored.
	|	TAG_TRANSFORM_TABLE size < TRANSFORM LIST>	The <TRANSFORM TABLE> starts with the TAG_TRANSFORM_TABLE tag. It is then followed by the encoded size field, in bytes, of the entire list of transforms.
< TRANSFORM LIST>	::=	< TRANSFORM BLOCK>	A < TRANSFORM LIST> must contain at least one entry.
	|	< TRANSFORM BLOCK> < TRANSFORM LIST>	
< TRANSFORM BLOCK>	::=	TAG_TRANSFORM M11, M12, M21, M22, DX, DY	General transform.
	|	TAG_TRANSFORM_ISOTROPIC_SCALE S	M11 = M22 = S, M12 =0, M21=0, DX = DY = 0
	|	TAG_TRANSFORM_ANISOTROPIC_SCLE M11, M22	M11 != M22, M12=0, M21=0, DX = DY = 0
	|	TAG_TRANSFORM_TRANSLATE DX, DY	M11=M12=M21=M22=0
	|	TAG_TRANSFORM_ROTATE A	M11 = M22 = cos(A), M12 = - M21 = sin(A), DX = DY = 0

	|	TAG_TRANSFORM_SCALE_AND_TRANSLATE M11, M22, DX, DY	M12 = M21 = 0
<BIT ASSIGNMENT TABLE>	::=	cBAB	Count of BAB blocks in the list, can be zero, in which case no BAB blocks follow
	|	<BAB LIST>	List of BAB blocks
<BAB LIST>	::=	<NULL>	
	|	<BAB BLOCK>	
	|	<BAB BLOCK> <BAB LIST>	
<BAB BLOCK>	::=	cEntries <bit packed array, 5 bits per entry>	(cEntries * 5 + 7)/8 bytes in the array, total
<INDEX_MAP_TABLE>	::=	cIndexMaps <IMB LIST>	Count of Index Map Blocks, may be zero
<IMB LIST>	::=	NULL	
	|	<IMB> <IMB LIST>	Index Map Block, potentially followed by more Index Map Blocks
<IMB>	::=	cEntries iBAB <DELTA-DELTA LIST >	MBE encoded count of entries, iBAB, index into BAB table, first 8 entries implicit, i.e. defined in the code, followed by cEntries long list of MBE encoded numbers that represent absolute delta-delta values for some packet property, listed in order of descending probabilities of occurrence.
<DELTA-DELTA LIST>	::=	<DELTA-DELTA>	MBE encoded number that represents an absolute delta-delta value that occurs with highest probability, followed by those that occur with smaller probability, if any.
	|	<DELTA-DELTA> <DELTA-DELTA LIST>	
<LOCAL INK PROPERTIES>	::=	NULL	The simplest ISF stream may contain no strokes.

	|	<PROPERTY LIST> <STROKE DESCRIPTOR INDEX> <DRAWING ATTRIBUTE INDEX> <TRANSFORM INDEX> <METRICS INDEX> <STROKE> <LOCAL INK PROPERTIES>	Or may contain only <PROPERTY_LIST> or only <STROKES> or some combination. < PROPERTY_LIST> are properties that may appear in the <WISP BODY> and out side of a <STROKE>. These are usually custom properties are rarely found in a WISP stream.
<PROPERTY LIST>	::=	NULL	
	|	GUID_IDX [size] value < PROPERTY LIST>	GUID_IDX can point to a predefined or custom GUID. [size] is omitted for some predefined GUIDs with known fixed size data types. When size is present it does not include algorithm byte.
<STROKE DESCRIPTOR INDEX>	::=	NULL	If there is no stroke descriptor index, 0 value is assumed.
	|	TAG_SIDX value	Or an index into the stroke descriptor table, which specifies the descriptor to use for the following STROKES. The index applies to all strokes that follow until the next TAG_SIDX.
<METRICS INDEX>	::=	NULL	If there is no Metrics index, 0 value is assumed.
	|	TAG_MIDX value	Or an index into the Metrics table, which specifies the descriptor to use for the following STROKES. The index applies to all strokes that follow until the next TAG_MIDX.
<DRAWING ATTRIBUTE INDEX>	::=	NULL	If there is no drawing attribute index, 0 value is assumed.
	|	TAG_DIDX value	Or an index into the Drawing Attribute Table, which specifies the Drawing Attribute Block to use for the following STROKES. The Drawing Attribute Index applies to all strokes that follow until the next TAG_DIDX.
< TRANSFORM INDEX>	::=	NULL	If there is no TRANSFORM index, 0 value is assumed.
	|	TAG_TIDX Value	Or an index into the TRANSFORM Table, which specifies the TRANSFORM Block to use for the following STROKES. The TRANSFORM Index applies to all strokes that follow until the next TAG_TIDX.
<STROKE>	::=	TAG_STROKE size	A stroke begins with the TAG_STROKE and is immediately followed by the encoded size, encoded number of points cPoints, <POINT_DATA>

		cPoints <POINT_DATA> <STROKE PROPERTIES>	and <STROKE PROPERTIES> cPoints is set to zero if there is no <POINT DATA>.
<POINT_DATA>	::=	NULL	<POINT_DATA> is defined by the stroke descriptor.
		Compression Algorithm ID, [size], Compressed. data <POINT_DATA>	Each entry consists of a byte indicating the compression algorithm, an optional byte-encoded size, which is only required for the compression types that are currently NOT defined in compress.h, and the compressed data itself. Each entry corresponds to an entry in the stroke descriptor.
<STROKE PROPERTIES>	::=	NULL	<STROKE PROPERTIES> are other properties associated with the stroke.
	|	<POINT PROPERTY BLOCK> <STROKE PROPERTIES>	Per point properties attach properties to a particular point in the stroke, such as comments, etc.
	|	GUID_IDX**, [**size**],** DATA <STROKE PROPERTIES>	Or the stroke may contain custom stroke properties. The [size] field is omitted for GUID_IDX values that correspond to predefined GUIDS with known fixed length data types. When size is present it does not include algorithm byte.
<POINT PROPERTY BLOCK>	::=	TAG_POINT_PROPERTY, size <POINT PROPERTY LIST>	The point property lists properties that are attached to several given points in the stroke. The encoded size field specifies the size of the whole <POINT PROPERTY LIST>
<POINT PROPERTY LIST>	::=	NULL	
	|	GUID_IDX, point index, size, data <POINT PROPERTY LIST>	Every entry contains GUID_IDX, point index of the point that this property is attached to, size of the data not counting the algorithm byte, and the data itself followed by more entries of this type.

Appendix

PREDEFINED TAGS

The following is a list of predefined tags used in this document:

TAG_INK_SPACE_RECT
TAG_COMPRESSION_MODE
TAG_GUID_TABLE
TAG_DRAW_ATTRS_TABLE
TAG_DRAW_ATTRS_BLOCK
TAG_DIDX
TAG_STROKE_DESC_TABLE
TAG_STROKE_DESC_BLOCK
TAG_SIDX
TAG_BUTTONS
TAG_NO_X
TAG_NO_Y
TAG_STROKE
TAG_STROKE_PROPERTY_LIST
TAG_POINT_PROPERTY
TAG_TRANSFORM_TABLE
TAG_TRANSFORM
TAG_TIDX
TAG_METRICS_TABLE
TAG_METRICS_BLOCK
TAG_MIDX
TAG_METRICS_MINIMUM
TAG_METRICS_MAXIMUM
TAG_METRICS_PERCISION
TAG_METRICS_PERCISION_UNITS
TAG_METRICS_UNIT_DIVISOR

```
#define     TAG_KNOWN_TAG_COUNT     50
```

In addition to these we define tags/indices for all of the predefined property GUIDs. All of these tags/indices have values in the range [0,99] as discussed above. The first TAG_KNOWN_TAG_COUNT indices are reserve for the above tags and few new ones that may be needed in the future. The indices in range [TAG_KNOWN_TAG_COUNT, 99] are reserved for predefined property GUIDs and the range >= 100 for the custom GUIDs.

This is the list of predefined packet property GUIDs is:

```
GUID_X,
GUID_Y,
GUID_Z,
GUID_PACKET_STATUS,
GUID_TIMER_TICK,            // 32 bit timer tick
GUID_SERIAL_NUMBER,
GUID_NORMAL_PRESSURE,
GUID_TANGENT_PRESSURE,
GUID_BUTTON_PRESSURE,
```

GUID_X_TILT_ORIENTATION,
GUID_Y_TILT_ORIENTATION,
GUID_AZIMUTH_ORIENTATION,
GUID_ALTITUDE_ORIENTATION,
GUID_TWIST_ORIENTATION,
GUID_PITCH_ROTATION,
GUID_ROLL_ROTATION,
GUID_YAW_ROTATION,

This is a list of predefined drawing attribute GUIDs

GUID_PEN_STYLE,
GUID_COLORREF,
GUID_PEN_WIDTH,
GUID_PEN_HEIGHT,
GUID_PEN_TIP,
GUID_DRAWING_FLAGS,
GUID_EDGE_SMOOTHING,

This is a list of GUIDs used by Microsoft Word

GUID_CHAR_ALTERNATES,
GUID_WORD_ALTERNATES,
GUID_GUIDE_STRUCTURE,
GUID_INKMETRICS,

This is a list of global ink property GUIDs

GUID_TIME_STAMP // stored as 64 bit value

MICROSOFT WORD EXAMPLE

0	WISP Version number, set to zero
cbInkObject	Size of the whole object and all its children
TAG_INK_SPACE_RECTANGLE	Word used the rectangle Width X 64K
Left, Top, Right, Bottom	Encoded Signed Values
TAG_DRAW_ATTRS_TABLE	Tag for the drawing attributes table
cbDrawingAttributesTable	Size of the drawing attributes table, the table only one entry
cbDrawingAttributeBlock	The size of the only drawing attribute block
TAG_PEN_WIDTH	Tag for the pen width
3	Value for the pen width, 3 in this example
TAG_COLORREF	Tag for the color
Encoded value for BLUE	Encoded value for color, e.g. BLUE
TAG_ALTERNATE_LIST	List of alternates for this word
cbAlternates	Size of all the alternates
Data for alternates	Compressed
TAG_INKMETRICS	Tag for the INKMETRICS data structure
Data for ink metrics	Byte encoded values, size is NOT needed
TAG_STROKE	Tag for the stroke
cbStroke	Size for stroke and all its children
cPoints	Count of points in this stroke
X data	Compressed X coordinates
Y data	Compressed Y coordinates
TAG_STROKE	Tag for stroke object
cbStroke	Size for stroke and all its children
cPoints	Count of Points in this stroke
X data	Compressed X coordinates
Y data	Compressed Y coordinates

MS Word breaks the ink into words and stores each word as a separate ISF. In the example above the ink object has 2 strokes. The strokes only have x,y arrays but no additional packet properties such as pressure. This is because the MS Word uses simple mouse input using the GetMouseMoveEx API. This API only reports x,y coordinates, but it reports no additional packet properties such as pressure. That is why the Stroke Descriptor Table is not needed. GetMouseMoveEx returns high precision x,y coordinates in absolute mode, scaled to a 64K x 64K rectangle. Word subsequently scales out X coordinates so as to match the aspect ratio of the screen on which the ink is displayed. For example on a 1024 x 768 pixel screen, the coordinates are scaled to the Ink Space Rectangle that is approximately 87K x 64K.

MS Word stores with each word the list of alternates that a recognizer returned for this ink object. Alternate list is variable size depending on the recognizer used and the ink itself, therefore, the size cbAlternates needs to be encoded in the stream, followed by the compressed list of alternate words. The non-compressed list of alternate words is simply a list of Unicode strings that are separated by zeros and zero terminated.

Likewise, MS Word stores the INKMETRICS data structure in the ink stream. This is a known fixed size data structure, so its size does not need to be encoded in the stream. Simply, the members of the structure are listed, one after another as byte-encoded values.

The Alternate List and INKMETRICS are examples of global ink properties stored in the ink stream.

Finally, we need to discuss the Drawing Attributes Table. It only consists of a single Drawing Attribute Block. This block applies to all the strokes; therefore it is not necessary to specify which Drawing Attribute Block applies to which stroke. This Drawing Attribute Block has two properties specified, the Pen Width of 3 and the Color blue. All other drawing attributes that are not mentioned in the Drawing Attribute Block are assumed to take on the default values. For example, the Pen Tip is assumed to be PEN_TIP_BALL.

cBits-cPads Lookup Table

All possible cBits,cPads combinations are listed. First 24 entries apply to bit packing of BYTE array, first 32 entries apply to bit packing of WORD array, and the whole table applies to bit packing of LONG array. There are at most 7 unused bits in the last byte of the compressed array. Therefore, given the cBits, the largest possible value for cPads is 7/ cBits.

Table Index	cBits	cPads
0	8	0
1	1	0
2	1	1
3	1	2
4	1	3
5	1	4
6	1	5
7	1	6
8	1	7
9	2	0
10	2	1
11	2	2
12	2	3
13	3	0
14	3	1
15	3	2
16	4	0
17	4	1
18	5	0
19	5	1
20	6	0
21	6	1
22	7	0
23	7	1
24	8	0
25	9	0
26	10	0
27	11	0
28	12	0
29	13	0
30	14	0
31	15	0
32	16	0
33	17	0
34	18	0
35	19	0
36	20	0
37	21	0
38	22	0
39	23	0
40	24	0
41	25	0
42	26	0
43	27	0
44	28	0
45	29	0
46	30	0
47	31	0

Appendix A: Tags

```
#define INDEX_GUID_X                        0
#define INDEX_GUID_Y                        1
#define INDEX_GUID_Z                        2
#define INDEX_GUID_PACKET_STATUS            3
#define INDEX_GUID_TIMER_TICK               4
#define INDEX_GUID_SERIAL_NUMBER            5
#define INDEX_GUID_NORMAL_PRESSURE          6
#define INDEX_GUID_TANGENT_PRESSURE         7
#define INDEX_GUID_BUTTON_PRESSURE          8
#define INDEX_GUID_X_TILT_ORIENTATION       9
#define INDEX_GUID_Y_TILT_ORIENTATION       10
#define INDEX_GUID_AZIMUTH_ORIENTATION      11
#define INDEX_GUID_ALTITUDE_ORIENTATION     12
#define INDEX_GUID_TWIST_ORIENTATION        13
#define INDEX_GUID_PITCH_ROTATION           14
#define INDEX_GUID_ROLL_ROTATION            15
#define INDEX_GUID_YAW_ROTATION             16
#define INDEX_GUID_PEN_STYLE                17
#define INDEX_GUID_COLORREF                 18
#define INDEX_GUID_PEN_WIDTH                19
#define INDEX_GUID_PEN_HEIGHT               20
#define INDEX_GUID_PEN_TIP                  21
#define INDEX_GUID_DRAWING_FLAGS            22
#define INDEX_GUID_CURSORID                 23
#define INDEX_GUID_WORD_ALTERNATES          24
#define INDEX_GUID_CHAR_ALTERNATES          25
#define INDEX_GUID_INKMETRICS               26
#define INDEX_GUID_GUIDE_STRUCTURE          27
#define INDEX_GUID_TIME_STAMP               28
#define INDEX_GUID_LANGUAGE                 29
#define INDEX_GUID_TRANSPARENCY             30
#define INDEX_GUID_CURVE_FITTING_ERROR      31
#define INDEX_GUID_RECO_LATTICE             32
#define INDEX_GUID_CURSORDOWN               33
#define INDEX_GUID_SECONDARYTIPSWITCH       34
#define INDEX_GUID_BARRELDOWN               35
#define INDEX_GUID_TABLETPICK               36
#define INDEX_GUID_ROP                              37

#define INDEX_GUID_MAX                      37

 const BYTE TAG_INK_SPACE_RECT                  = 0;
 const BYTE TAG_GUID_TABLE                      = 1;
 const BYTE TAG_DRAW_ATTRS_TABLE                = 2;
 const BYTE TAG_DRAW_ATTRS_BLOCK                = 3;
 const BYTE TAG_STROKE_DESC_TABLE               = 4;
 const BYTE TAG_STROKE_DESC_BLOCK               = 5;
 const BYTE TAG_BUTTONS                         = 6;
 const BYTE TAG_NO_X                            = 7;
 const BYTE TAG_NO_Y                            = 8;
 const BYTE TAG_DIDX                            = 9;
 const BYTE TAG_STROKE                          = 10;
 const BYTE TAG_STROKE_PROPERTY_LIST            = 11;
```

```
 const BYTE TAG_POINT_PROPERTY                  = 12;
 const BYTE TAG_SIDX                            = 13;
 const BYTE TAG_COMPRESSION_HEADER              = 14;
 const BYTE TAG_TRANSFORM_TABLE                 = 15;
 const BYTE TAG_TRANSFORM                       = 16;
 const BYTE TAG_TRANSFORM_ISOTROPIC_SCALE       = 17;
 const BYTE TAG_TRANSFORM_ANISOTROPIC_SCALE     = 18;
 const BYTE TAG_TRANSFORM_ROTATE                = 19;
 const BYTE TAG_TRANSFORM_TRANSLATE             = 20;
 const BYTE TAG_TRANSFORM_SCALE_AND_TRANSLATE   = 21;
 const BYTE TAG_TRANSFORM_QUAD                  = 22;
 const BYTE TAG_TIDX                            = 23;
 const BYTE TAG_METRIC_TABLE                    = 24;
 const BYTE TAG_METRIC_BLOCK                    = 25;
 const BYTE TAG_MIDX                            = 26;
 const BYTE TAG_MANTISSA                        = 27;
 const BYTE TAG_PERSISTENT_FORMAT                     = 28;
 const BYTE TAG_HIMETRIC_SIZE                         = 29;
 const BYTE TAG_STROKE_IDS                      = 30;

#define MAX_KNOWN_TAG_COUNT     50

 const BYTE TAG_KNOWN_TAG_COUNT = MAX_KNOWN_TAG_COUNT;

 const ULONG KNOWN_GUID_BASE_INDEX = MAX_KNOWN_TAG_COUNT;

#define MAX_KNOWN_GUID_INDEX    100

 const ULONG KNOWN_GUID_INDEX_LIMIT = MAX_KNOWN_GUID_INDEX;
 const ULONG CUSTOM_GUID_BASE_INDEX = MAX_KNOWN_GUID_INDEX;

 const GUID FAR KNOWN_GUIDS[38] =
{

    { 0x598a6a8f, 0x52c0, 0x4ba0, { 0x93, 0xaf, 0xaf, 0x35, 0x74, 0x11, 0xa5,
0x61 } },

    { 0xb53f9f75, 0x04e0, 0x4498, { 0xa7, 0xee, 0xc3, 0x0d, 0xbb, 0x5a, 0x90,
0x11 } },

    { 0x735adb30, 0x0ebb, 0x4788, { 0xa0, 0xe4, 0x0f, 0x31, 0x64, 0x90, 0x05,
0x5d } },

    { 0x6e0e07bf, 0xafe7, 0x4cf7, { 0x87, 0xd1, 0xaf, 0x64, 0x46, 0x20, 0x84,
0x18 } },

    { 0x436510c5, 0xfed3, 0x45d1, { 0x8b, 0x76, 0x71, 0xd3, 0xea, 0x7a, 0x82,
0x9d } },

    { 0x78a81b56, 0x0935, 0x4493, { 0xba, 0xae, 0x00, 0x54, 0x1a, 0x8a, 0x16,
0xc4 } },

    { 0x7307502d, 0xf9f4, 0x4e18, { 0xb3, 0xf2, 0x2c, 0xe1, 0xb1, 0xa3, 0x61,
0x0c } },

    { 0x6da4488b, 0x5244, 0x41ec, { 0x90, 0x5b, 0x32, 0xd8, 0x9a, 0xb8, 0x08,
0x09 } },
```

```
    { 0x8b7fefc4, 0x96aa, 0x4bfe, { 0xac, 0x26, 0x8a, 0x5f, 0x0b, 0xe0, 0x7b,
0xf5 } },

    { 0xa8d07b3a, 0x8bf0, 0x40b0, { 0x95, 0xa9, 0xb8, 0x0a, 0x6b, 0xb7, 0x87,
0xbf } },

    { 0x0e932389, 0x1d77, 0x43af, { 0xac, 0x00, 0x5b, 0x95, 0x0d, 0x6d, 0x4b,
0x2d } },

    { 0x029123b4, 0x8828, 0x410b, { 0xb2, 0x50, 0xa0, 0x53, 0x65, 0x95, 0xe5,
0xdc } },

    { 0x82dec5c7, 0xf6ba, 0x4906, { 0x89, 0x4f, 0x66, 0xd6, 0x8d, 0xfc, 0x45,
0x6c } },

    { 0x0d324960, 0x13b2, 0x41e4, { 0xac, 0xe6, 0x7a, 0xe9, 0xd4, 0x3d, 0x2d,
0x3b } },

    { 0x7f7e57b7, 0xbe37, 0x4be1, { 0xa3, 0x56, 0x7a, 0x84, 0x16, 0x0e, 0x18,
0x93 } },

    { 0x5d5d5e56, 0x6ba9, 0x4c5b, { 0x9f, 0xb0, 0x85, 0x1c, 0x91, 0x71, 0x4e,
0x56 } },

    { 0x6a849980, 0x7c3a, 0x45b7, { 0xaa, 0x82, 0x90, 0xa2, 0x62, 0x95, 0x0e,
0x89 } },

    { 0x33c1df83, 0xecdb, 0x44f0, { 0xb9, 0x23, 0xdb, 0xd1, 0xa5, 0xb2, 0x13,
0x6e } },

    { 0x5329cda5, 0xfa5b, 0x4ed2, { 0xbb, 0x32, 0x83, 0x46, 0x01, 0x72, 0x44,
0x28 } },

    { 0x002df9af, 0xdd8c, 0x4949, { 0xba, 0x46, 0xd6, 0x5e, 0x10, 0x7d, 0x1a,
0x8a } },

    { 0x9d32b7ca, 0x1213, 0x4f54, { 0xb7, 0xe4, 0xc9, 0x05, 0x0e, 0xe1, 0x7a,
0x38 } },

    { 0xe71caab9, 0x8059, 0x4c0d, { 0xa2, 0xdb, 0x7c, 0x79, 0x54, 0x47, 0x8d,
0x82 } },

    { 0x5c0b730a, 0xf394, 0x4961, { 0xa9, 0x33, 0x37, 0xc4, 0x34, 0xf4, 0xb7,
0xeb } },

    { 0x2812210f, 0x871e, 0x4d91, { 0x86, 0x07, 0x49, 0x32, 0x7d, 0xdf, 0x0a,
0x9f } },

    { 0x8359a0fa, 0x2f44, 0x4de6, { 0x92, 0x81, 0xce, 0x5a, 0x89, 0x9c, 0xf5,
0x8f } },

    { 0x4c4642dd, 0x479e, 0x4c66, { 0xb4, 0x40, 0x1f, 0xcd, 0x83, 0x95, 0x8f,
0x00 } },

    { 0xce2d9a8a, 0xe58e, 0x40ba, { 0x93, 0xfa, 0x18, 0x9b, 0xb3, 0x90, 0x00,
0xae } },
```

```
    { 0xc3c7480f, 0x5839, 0x46ef, { 0xa5, 0x66, 0xd8, 0x48, 0x1c, 0x7a, 0xfe,
0xc1 } },

    { 0xea2278af, 0xc59d, 0x4ef4, { 0x98, 0x5b, 0xd4, 0xbe, 0x12, 0xdf, 0x22,
0x34 } },

    { 0xb8630dc9, 0xcc5c, 0x4c33, { 0x8d, 0xad, 0xb4, 0x7f, 0x62, 0x2b, 0x8c,
0x79 } },

    { 0x15e2f8e6, 0x6381, 0x4e8b, { 0xa9, 0x65, 0x01, 0x1f, 0x7d, 0x7f, 0xca,
0x38 } },

    { 0x7066fbe4, 0x473e, 0x4675, { 0x9c, 0x25, 0x00, 0x26, 0x82, 0x9b, 0x40,
0x1f } },

    { 0xbbc85b9a, 0xade6, 0x4093, { 0xb3, 0xbb, 0x64, 0x1f, 0xa1, 0xd3, 0x7a,
0x1a } },

    { 0x39143d3,  0x78cb, 0x449c, { 0xa8, 0xe7, 0x67, 0xd1, 0x88, 0x64, 0xc3,
0x32 } },

    { 0x67743782, 0xee5,  0x419a, { 0xa1, 0x2b, 0x27, 0x3a, 0x9e, 0xc0, 0x8f,
0x3d } },

    { 0xf0720328, 0x663b, 0x418f, { 0x85, 0xa6, 0x95, 0x31, 0xae, 0x3e, 0xcd,
0xfa } },

    { 0xa1718cdd, 0xdac,  0x4095, { 0xa1, 0x81, 0x7b, 0x59, 0xcb, 0x10, 0x6b,
0xfb } },

      { 0x810a74d2, 0x6ee2, 0x4e39, { 0x82, 0x5e, 0x6d, 0xef, 0x82, 0x6a,
0xff, 0xc5 } },

};

 const ULONG KNOWN_GUID_COUNT = sizeof(KNOWN_GUIDS) / sizeof(GUID);

 const GUID& GUID_X                      = KNOWN_GUIDS[INDEX_GUID_X
];
 const GUID& GUID_Y                      = KNOWN_GUIDS[INDEX_GUID_Y
];
 const GUID& GUID_Z                      = KNOWN_GUIDS[INDEX_GUID_Z
];
 const GUID& GUID_PACKET_STATUS          =
KNOWN_GUIDS[INDEX_GUID_PACKET_STATUS       ];
 const GUID& GUID_TIMER_TICK             = KNOWN_GUIDS[INDEX_GUID_TIMER_TICK
];
 const GUID& GUID_SERIAL_NUMBER          =
KNOWN_GUIDS[INDEX_GUID_SERIAL_NUMBER       ];
 const GUID& GUID_NORMAL_PRESSURE        =
KNOWN_GUIDS[INDEX_GUID_NORMAL_PRESSURE     ];
 const GUID& GUID_TANGENT_PRESSURE       =
KNOWN_GUIDS[INDEX_GUID_TANGENT_PRESSURE    ];
 const GUID& GUID_BUTTON_PRESSURE        =
KNOWN_GUIDS[INDEX_GUID_BUTTON_PRESSURE     ];
```

```
 const GUID& GUID_X_TILT_ORIENTATION      =
KNOWN_GUIDS[INDEX_GUID_X_TILT_ORIENTATION  ];
 const GUID& GUID_Y_TILT_ORIENTATION      =
KNOWN_GUIDS[INDEX_GUID_Y_TILT_ORIENTATION  ];
 const GUID& GUID_AZIMUTH_ORIENTATION     =
KNOWN_GUIDS[INDEX_GUID_AZIMUTH_ORIENTATION ];
 const GUID& GUID_ALTITUDE_ORIENTATION    =
KNOWN_GUIDS[INDEX_GUID_ALTITUDE_ORIENTATION];
 const GUID& GUID_TWIST_ORIENTATION       =
KNOWN_GUIDS[INDEX_GUID_TWIST_ORIENTATION   ];
 const GUID& GUID_PITCH_ROTATION          =
KNOWN_GUIDS[INDEX_GUID_PITCH_ROTATION      ];
 const GUID& GUID_ROLL_ROTATION           =
KNOWN_GUIDS[INDEX_GUID_ROLL_ROTATION       ];
 const GUID& GUID_YAW_ROTATION            =
KNOWN_GUIDS[INDEX_GUID_YAW_ROTATION        ];
 const GUID& GUID_PEN_STYLE               = KNOWN_GUIDS[INDEX_GUID_PEN_STYLE
];
 const GUID& GUID_COLORREF                = KNOWN_GUIDS[INDEX_GUID_COLORREF
];
 const GUID& GUID_PEN_WIDTH               = KNOWN_GUIDS[INDEX_GUID_PEN_WIDTH
];
 const GUID& GUID_PEN_HEIGHT              = KNOWN_GUIDS[INDEX_GUID_PEN_HEIGHT
];
 const GUID& GUID_PEN_TIP                 = KNOWN_GUIDS[INDEX_GUID_PEN_TIP
];
 const GUID& GUID_DRAWING_FLAGS           =
KNOWN_GUIDS[INDEX_GUID_DRAWING_FLAGS       ];
 const GUID& GUID_CURSORID                = KNOWN_GUIDS[INDEX_GUID_CURSORID
];
 const GUID& GUID_WORD_ALTERNATES         =
KNOWN_GUIDS[INDEX_GUID_WORD_ALTERNATES     ];
 const GUID& GUID_CHAR_ALTERNATES         =
KNOWN_GUIDS[INDEX_GUID_CHAR_ALTERNATES     ];
 const GUID& GUID_INKMETRICS              = KNOWN_GUIDS[INDEX_GUID_INKMETRICS
];
 const GUID& GUID_GUIDE_STRUCTURE         =
KNOWN_GUIDS[INDEX_GUID_GUIDE_STRUCTURE     ];
 const GUID& GUID_TIME_STAMP              = KNOWN_GUIDS[INDEX_GUID_TIME_STAMP
];
 const GUID& GUID_LANGUAGE                = KNOWN_GUIDS[INDEX_GUID_LANGUAGE
];
 const GUID& GUID_TRANSPARENCY            =
KNOWN_GUIDS[INDEX_GUID_TRANSPARENCY        ];
 const GUID& GUID_CURVE_FITTING_ERROR     =
KNOWN_GUIDS[INDEX_GUID_CURVE_FITTING_ERROR ];
 const GUID& GUID_RECO_LATTICE            =
KNOWN_GUIDS[INDEX_GUID_RECO_LATTICE        ];

 const GUID& GUID_CURSORDOWN              = KNOWN_GUIDS[INDEX_GUID_CURSORDOWN
];
 const GUID& GUID_SECONDARYTIPSWITCH      =
KNOWN_GUIDS[INDEX_GUID_SECONDARYTIPSWITCH  ];
 const GUID& GUID_BARRELDOWN              = KNOWN_GUIDS[INDEX_GUID_BARRELDOWN
];
 const GUID& GUID_TABLETPICK              = KNOWN_GUIDS[INDEX_GUID_TABLETPICK
];
```

```
 const GUID& GUID_ROP                                =
KNOWN_GUIDS[INDEX_GUID_ROP                           ];

#define GUID_X                              (*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_X
]))))
#define GUID_Y                              (*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_Y
]))))
#define GUID_Z                              (*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_Z
]))))
#define GUID_PACKET_STATUS
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_PACKET_STATUS       ])))
#define GUID_TIMER_TICK
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_TIMER_TICK          ])))
#define GUID_SERIAL_NUMBER
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_SERIAL_NUMBER       ])))
#define GUID_NORMAL_PRESSURE
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_NORMAL_PRESSURE     ])))
#define GUID_TANGENT_PRESSURE
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_TANGENT_PRESSURE    ])))
#define GUID_BUTTON_PRESSURE
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_BUTTON_PRESSURE     ])))
#define GUID_X_TILT_ORIENTATION
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_X_TILT_ORIENTATION  ])))
#define GUID_Y_TILT_ORIENTATION
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_Y_TILT_ORIENTATION  ])))
#define GUID_AZIMUTH_ORIENTATION
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_AZIMUTH_ORIENTATION ])))
#define GUID_ALTITUDE_ORIENTATION
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_ALTITUDE_ORIENTATION])))
#define GUID_TWIST_ORIENTATION
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_TWIST_ORIENTATION   ])))
#define GUID_PITCH_ROTATION
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_PITCH_ROTATION      ])))
#define GUID_ROLL_ROTATION
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_ROLL_ROTATION       ])))
#define GUID_YAW_ROTATION
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_YAW_ROTATION        ])))
#define GUID_PEN_STYLE
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_PEN_STYLE           ])))
#define GUID_COLORREF
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_COLORREF            ])))
#define GUID_PEN_WIDTH
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_PEN_WIDTH           ])))
#define GUID_PEN_HEIGHT
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_PEN_HEIGHT          ])))
#define GUID_PEN_TIP
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_PEN_TIP             ])))
#define GUID_DRAWING_FLAGS
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_DRAWING_FLAGS       ])))
#define GUID_CURSORID
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_CURSORID            ])))
#define GUID_WORD_ALTERNATES
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_WORD_ALTERNATES     ])))
#define GUID_CHAR_ALTERNATES
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_CHAR_ALTERNATES     ])))
#define GUID_INKMETRICS
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_INKMETRICS          ])))
```

```
#define GUID_GUIDE_STRUCTURE
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_GUIDE_STRUCTURE     ])))
#define GUID_TIME_STAMP
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_TIME_STAMP          ])))
#define GUID_LANGUAGE
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_LANGUAGE            ])))
#define GUID_TRANSPARENCY
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_TRANSPARENCY        ])))
#define GUID_CURVE_FITTING_ERROR
(*((GUID*)&(KNOWN_GUIDS[INDEX_GUID_CURVE_FITTING_ERROR ])))
#define GUID_RECO_LATTICE               (*((GUID*)&(
KNOWN_GUIDS[INDEX_GUID_RECO_LATTICE       ])))

#define GUID_CURSORDOWN                 (*((GUID*)&(
KNOWN_GUIDS[INDEX_GUID_CURSORDOWN         ])))
#define GUID_SECONDARYTIPSWITCH         (*((GUID*)&(
KNOWN_GUIDS[INDEX_GUID_SECONDARYTIPSWITCH ])))
#define GUID_BARRELDOWN                 (*((GUID*)&(
KNOWN_GUIDS[INDEX_GUID_BARRELDOWN         ])))
#define GUID_TABLETPICK                 (*((GUID*)&(
KNOWN_GUIDS[INDEX_GUID_TABLETPICK         ])))
#define GUID_ROP                                (*((GUID*)&(
KNOWN_GUIDS[INDEX_GUID_ROP                        ])))

DEFINE_GUID(GUID_LINENUMBER, 0xdbf29f2c, 0x5289, 0x4be8, 0xb3, 0xd8, 0x6e,
0xf6, 0x32, 0x46, 0x25, 0x3e);
DEFINE_GUID(GUID_BOXNUMBER,  0x2c243e3a, 0xf733, 0x4eb6, 0xb1, 0xf8, 0xb5,
0xdc, 0x5c, 0x2c, 0x4c, 0xda);
DEFINE_GUID(GUID_SEGMENTATION, 0xb3c0fe6c, 0xfb51, 0x4164, 0xba, 0x2f, 0x84,
0x4a, 0xf8, 0xf9, 0x83, 0xda);
// {CA6F40DC-5292-452a-91FB-2181C0BEC0DE}
DEFINE_GUID(GUID_HOTPOINT, 0xca6f40dc, 0x5292, 0x452a, 0x91, 0xfb, 0x21,
0x81, 0xc0, 0xbe, 0xc0, 0xde);
// {BF0EEC4E-4B7D-47a9-8CFA-234DD24BD22A}
DEFINE_GUID(GUID_MAX_STROKE_COUNT, 0xbf0eec4e, 0x4b7d, 0x47a9, 0x8c, 0xfa,
0x23, 0x4d, 0xd2, 0x4b, 0xd2, 0x2a);
// {7DFE11A7-FB5D-4958-8765-154ADF0D833F}
DEFINE_GUID(GUID_CONFIDENCELEVEL, 0x7dfe11a7, 0xfb5d, 0x4958, 0x87, 0x65,
0x15, 0x4a, 0xdf, 0xd, 0x83, 0x3f);
// {8CC24B27-30A9-4b96-9056-2D3A90DA0727}
DEFINE_GUID(GUID_LINEMETRICS, 0x8cc24b27, 0x30a9, 0x4b96, 0x90, 0x56, 0x2d,
0x3a, 0x90, 0xda, 0x7, 0x27);

DEFINE_GUID(GUID_INRANGE, 0xdc00b1af, 0x7321, 0x4ac1, 0x91, 0x88, 0xe3, 0x20,
0x18, 0xeb, 0xb2, 0x3b);
DEFINE_GUID(GUID_TOUCH, 0x65c98c60, 0xcd80, 0x447d, 0xb1, 0x29, 0x25, 0xf6,
0xe, 0x1d, 0x80, 0x5b);
DEFINE_GUID(GUID_UNTOUCH, 0x378c85bb, 0x7118, 0x491e, 0x85, 0x16, 0xa7, 0x48,
0x2d, 0xb, 0x68, 0x3a);
DEFINE_GUID(GUID_TAP, 0x9eaad4, 0xd133, 0x4ed2, 0xb1, 0x2c, 0x89, 0x1f, 0x8e,
0x82, 0x5e, 0x6a);
DEFINE_GUID(GUID_QUALITY, 0xb7fe8008, 0x2df6, 0x4e1b, 0x8c, 0x43, 0xf5, 0xf1,
0x6, 0x8, 0x93, 0x2d);
DEFINE_GUID(GUID_DATAVALID, 0xaacf46b5, 0xf107, 0x47b0, 0xb5, 0x77, 0xd9,
0x39, 0xc0, 0x53, 0xed, 0x41);
DEFINE_GUID(GUID_TRANSDUCERINDEX, 0xa412b445, 0x7818, 0x4c83, 0x84, 0x55,
0x49, 0x29, 0xc8, 0x70, 0x76, 0x3b);
```

```
DEFINE_GUID(GUID_TABLETFUNCTIONKEYS, 0xff3b8afe, 0x5f06, 0x494d, 0xa4, 0xf8,
0xd3, 0xe2, 0x85, 0xf9, 0x76, 0x30);
DEFINE_GUID(GUID_PROGRAMCHANGEKEYS, 0x869c344a, 0x92a1, 0x4b6f, 0xb4, 0xbc,
0x3, 0x96, 0xc6, 0xa9, 0xf6, 0xaf);
DEFINE_GUID(GUID_BATTERYSTRENGTH, 0x4ca0a0dc, 0x3549, 0x43f5, 0xa0, 0x32,
0x99, 0xf4, 0xe3, 0x3d, 0xf4, 0x90);
DEFINE_GUID(GUID_INVERT, 0xc3aa28c8, 0x806b, 0x490c, 0xa8, 0x4a, 0xa7, 0x7a,
0xe7, 0x27, 0xc7, 0x18);
DEFINE_GUID(GUID_BUTTON4,0x844b06d, 0xaa2c, 0x4c66, 0x99, 0x76, 0x88, 0xdd,
0x2a, 0x59, 0xe4, 0xf0);
DEFINE_GUID(GUID_BUTTON5,0x944d1340, 0x2549, 0x4905, 0xbd, 0x54, 0x3e, 0xe3,
0x96, 0x3e, 0xe1, 0x57);
DEFINE_GUID(GUID_BUTTON6,0xff19bd41, 0xa463, 0x4eaa, 0xaf, 0x10, 0xb5, 0x6,
0x48, 0x79, 0xe5, 0x4b);
DEFINE_GUID(GUID_BUTTON7,0xdedaf13c, 0xb7dc, 0x423b, 0xb9, 0x7f, 0xa2, 0xbd,
0x68, 0xa2, 0xfd, 0x3d);
DEFINE_GUID(GUID_BUTTON8,0xa6f70e64, 0x3a67, 0x4552, 0xa0, 0xc4, 0x17, 0x38,
0x4e, 0x49, 0x5a, 0x55);
DEFINE_GUID(GUID_BUTTON9,0xbf55916c, 0xd6e6, 0x4fd3, 0x94, 0x62, 0x55, 0xd6,
0x9d, 0xb, 0xe7, 0x9c);
DEFINE_GUID(GUID_BUTTON10,0x95f1b222, 0x1159, 0x4f9a, 0xb6, 0xe4, 0x6e, 0xde,
0xdf, 0xc7, 0x56, 0x9b);
DEFINE_GUID(GUID_BUTTON11,0xf2dff7da, 0xf458, 0x4b61, 0xa2, 0x15, 0x62, 0xd0,
0x56, 0x48, 0x6, 0x53);
DEFINE_GUID(GUID_BUTTON12,0x6a860858, 0x9b68, 0x4da6, 0xb3, 0x2a, 0x55, 0xe9,
0xe9, 0x75, 0xbe, 0xeb);
DEFINE_GUID(GUID_BUTTON13,0x42ffb4d9, 0x7f95, 0x475e, 0x9c, 0x97, 0x82, 0xa0,
0x2b, 0xdc, 0x7e, 0xb6);
DEFINE_GUID(GUID_BUTTON14,0x215008c8, 0xf09d, 0x48d7, 0x95, 0x2e, 0xc, 0x11,
0x8d, 0x6, 0xe8, 0xca);
DEFINE_GUID(GUID_BUTTON15, 0xd1d1fa37, 0x1ee0, 0x4015, 0x9c, 0x33, 0x12,
0xcc, 0x42, 0x57, 0x60, 0x1);

DEFINE_GUID(GUID_PEN_TIMESTAMP1, 0x413a7d1a, 0xeede, 0x45ce, 0xa4, 0x32,
0x80, 0x88, 0xca, 0x9e, 0x8e, 0x4a);
DEFINE_GUID(GUID_PEN_TIMESTAMP2, 0x876c825, 0xcdc, 0x4395, 0xbf, 0x9b, 0x36,
0x7e, 0x69, 0x8a, 0x75, 0x56);

DEFINE_GUID(GUID_WIDTH, 0xbaabe94d, 0x2712, 0x48f5, 0xbe, 0x9d, 0x8f, 0x8b,
0x5e, 0xa0, 0x71, 0x1a);
DEFINE_GUID(GUID_HEIGHT, 0xe61858d2, 0xe447, 0x4218, 0x9d, 0x3f, 0x18, 0x86,
0x5c, 0x20, 0x3d, 0xf4);
DEFINE_GUID(GUID_FINGERCONTACTCONFIDENCE, 0xe706c804, 0x57f0, 0x4f00, 0x8a,
0x0c, 0x85, 0x3d, 0x57, 0x78, 0x9b, 0xe9);
DEFINE_GUID(GUID_TEMPID, 0x2585b91, 0x49b, 0x4750, 0x96, 0x15, 0xdf, 0x89,
0x48, 0xab, 0x3c, 0x9c);
```