12. Data Formats for Properties and Property Sets

OLE 2 defines a standard structure for a data format capable of representing groups of tagged data values, known as properties. The standard structure is generic with respect to the semantics of the properties being represented; it says nothing about what they mean. The standard allows for the ability to create extensions to sets over time without affecting extant clients of the base set.

Beyond defining a standard structure, OLE 2 defines just one standard property set, the Document Summary Information set. By storing this information in their storage in the manner outlined below, applications using Compound Files for their document file formats allow this information about the contents of their file to be read and understood by generic file browsing tools.[120]

12.1. What are Properties

Property sets are tagged collections of values, whose meaning is known to the code that manipulates them. That is to say, the meaning is known as far as that code needs to know it. Generic browsers that do not modify property sets may not need to know very much of the meaning (schema). Another way to think of property sets is as a convention for structuring data that can be manipulated or communicated by independent pieces of code, some of which are generic, and some of which have specific knowledge of a specific property set.

Examples of property sets are the character formatting properties in a word processor, or rendering attributes of an element in a drawing program.

Property sets may be stored (in IStorage/IStream instances) and communicated (through IDataObject and other interfaces such as the clipboard) in the representation described here. This provides a common representation that can be used for an open-ended set of property sets, and enables common code to deal with them. The representation is partially self-identifying.

All data elements are stored in Intel representation (byte order etc.).

12.2. Document Properties in Storage

Applications may choose to expose some of the state of their documents so that other applications can locate and read that information. Examples are a property set describing the author, title, and keywords of a document created with a word processor, or the list of fonts used in a document. This facility is not restricted to documents; it can also be used on embedded objects.

To expose a property set in this manner, the application creates an IStorage or IStream instance in the same level of the storage structure as its data streams. Names beginning with '\0x05' are reserved for applications to use for the property sets that they create on their documents for other applications to see. It follows that the application chooses the names, either from published names and formats, or by creating a new name, assigning a class identifier and format identifier, and publishing them if so desired. A property set may be stored in a single stream, or in an IStorage instance which contains multiple streams. In the latter case, the contained stream named "CONTENTS" is the primary stream containing property values, where some values may be names of other streams or IStorage instances within the storage for this property set.

For a property set that uses an IStorage instance, the application may set the CLSID of the IStorage to be the same as that stored in the contents stream (see below).

Property set formats are identified by unique identifiers, by analogy to class identifiers and interface identifiers, and these identifiers are represented and assigned in the same way. A property set may be defined as an extension of an existing property set, by allocating a new format ID, and allocating property IDs that

[120] This is an interim solution. It is currently our intention that longer term a handler-like architecture will be defined for files, enabling the same sort of interoperable capability for documents not choosing to use Compound Files.

do not conflict with the base set. The serialized representation includes the base set and the extension, and both format IDs. This allows readers to skip over unknown extensions, and preserve them, while still extracting information from the properties that they do know.

12.3. Serialized Format for Property Sets

The stream contains a header, a class identifier, a count of sections (which must be at least one), a table of format ID and offset for each section, and a series of sections. Each offset is the distance in bytes from the start of the whole stream where the section begins.

Byte-order indicator	Format version	Originating OS version	Class identifier	Count of sections	(Format ID, offset) pairs	Section (repeated)
WORD	WORD	DWORD	CLSID (16 bytes)	DWORD	count * (FMTID + DWORD)	variable
0xfffe	0					

The OS version number is encoded as OS kind in the high order word (0 for Windows on DOS, 1 for Macintosh, 2 for Windows NT) and the OS-supplied version number in the low order word. For Windows on DOS and Windows NT, the latter is the low order word of the result of `GetVersion()`.

The class identifier is that of a class that can display and/or provide programmatic access to the property values. If there is no such class, the application should set the class ID to be the same as the Format ID (which is explained in the next section).

A format identifier (FMTID) is represented as a DWORD, two WORDs, and 8 bytes, in sequence.

Sections are tagged with a format identifier. The format identifier is what determines the set of identifiers in the property set, their meaning, and any constraints on the values. In other words, if the format identifier was known to the author of some piece of code, then he knew how to manipulate this property set. To allow for extension of property sets over time, while still permitting older clients to access the properties that they do understand, the values are stored in a sequence of sections, each tagged with a format identifier. Each property set format identifier indicates the meaning of the properties in the section, and succeeding sections are assumed to be defined as extensions of the preceding one, so that property identifiers do not conflict. Property set format identifiers should be assigned using the same scheme as class identifiers and interface identifiers.

A section contains a byte count for the section, a count of the property values in the section, an array of 32bit property identifiers and offsets, and an array of property (type, value) pairs. These two arrays are in corresponding order. Offsets are the distance from the start of the section to the start of the property (type, value) pair. This allows sections to be copied as blobs.

count of bytes following in this section	count of properties	(property ID, offset) pairs	type	value	type, value repeated
DWORD	DWORD	count*(DWORD+DWORD)	DWORD	variable	

Property ID zero is reserved in all property sets for an optional dictionary giving human readable names for the property set itself, and for the properties in the set. The value will be an array of `count+1` (property ID, string) pairs, where the first ID is zero and its corresponding string is the name of the property set, and the remainder are the IDs and corresponding names of the properties. This dictionary may omit entries for properties that are assumed to be universally known by clients that manipulate the property set. Typically names for the base property sets for widely accepted standards will be omitted, but extensions or special purpose sets may include dictionaries for use by browsers etc.

Property ID one is reserved for a code page indicator, which identifies the code page for textual property values. All values must be stored with the same code page. If the code page indicator is not present, the prevailing code page on the reader's machine must be assumed.

Properties may be omitted from the stored set; readers must be robust in this case.

A property (type, value) pair is a DWORD type indicator, followed by a value whose representation depends on the type. Type indicators are and their associated values are defined in the header file variant.h which is part of the OLE2 developer's kit. The types that are valid in stored property sets are:

Type indicator	Code	Value Representation
VT_EMPTY	0	none
VT_NULL	1	none
VT_I2	2	WORD
VT_I4	3	DWORD
VT_R4	4	32bit IEEE Floating point
VT_R8	5	64bit IEEE Floating point
VT_CY	6	8 byte two's complement integer (scaled by 10,000), use for currency amounts
VT_DATE	7	Time format used by many applications: a 64bit floating point number representing seconds January 1st 1900. This is stored in the same representation as VT_R8
VT_BSTR	8	Counted, null terminated binary string; represented as a DWORD byte count (including the terminating null) followed by the bytes of data.
VT_BOOL	11	Boolean value, WORD containing 0 (false) or -1 (true). (N.B this will be zero padded to 32bit boundary.)
VT_VARIANT	12	a type indicator followed by the corresponding value. This is only used in conjunction with VT_VECTOR: see below.
VT_I8	20	8 byte signed integer
VT_LPSTR	30	same as VT_BSTR; this is the representation of most strings
VT_LPWSTR	31	A counted and null terminated Unicode string; a DWORD character count (where the count includes the terminating null) followed by that many Unicode (16bit) characters. Note that the count is not a byte count.
VT_FILETIME	64	64bit File time structure as defined by Win32
VT_BLOB	65	DWORD count of bytes, followed by that many bytes of data. This is similar to VT_BSTR but does not guarantee a null byte at the end of the data
VT_STREAM	66	this indicates the value is stored in a stream which is sibling to the CONTENTS stream Following this type indicator is a VT_LPSTR which names the stream containing the data
VT_STORAGE	67	this indicates the value is stored in an IStorage which is sibling to the CONTENTS stream Following this type indicator is a VT_LPSTR which names the IStorage containing the data
VT_STREAMED_OBJECT	68	as VT_STREAM but indicates that the stream contains a serialized object, which is a class ID followed by initialization data for the class.
VT_STORED_OBJECT	69	as VT_STORAGE but indicates that the IStorage contains an object
VT_BLOB_OBJECT	70	A BLOB containing a serialized object in the same representation as would appear in a VT_STREAMED_OBJECT. The only significant difference is that this type does not have the system-level storage overhead that VT_STREAMED_OBJECT would have, and is therefore more suitable for scenarios involving numbers of small objects.
VT_CF	71	A BLOB containing a clipboard format identifier followed by the data in that format.
VT_CLSID	72	a class ID, which is a DWORD, two WORDs, and 8 bytes
VT_VECTOR	0x1000	if the type indicator is one of the above values with this bit on in addition, then the value is a DWORD count of elements, followed by that many repetitions of the value. As an example, a type indicator of VT_LPSTR|VT_VECTOR has a DWORD element count, a DWORD byte count, the first string data, a DWORD byte count, the second string data, and so on.

All type/value pairs begin on a 32bit boundary, which means that in turn, type indicators and values are so aligned. This means that values may be followed by null bytes to align a subsequent pair.

A clipboard format tag for a VT_CF value is represented as follows. If the format value of the presentation data is one of the Windows pre-defined clipboard format values then the first field in the above structure will have -1L and the second field will contain the format number. For Macintosh four-byte clipboard format tags, the first field is -2L and second field is the tag. Otherwise the format of presentation data is normally registered via RegisterClipboardFormat. In this case the first field will be the length of the string and the second field will have the clipboard format name (zero terminated). There are some uses where the format name is empty. In the case that a property set format identifier is used, it is stored as -3L followed by 16 bytes. Thus there are five cases:

-1L	Windows Clipboard format value
Long	DWORD

-2L	Mac format value
Long	DWORD

-3L	format identifier
Long	DWORD, WORD, WORD, 8 bytes

Length of name	Clipboard format name
Long	Variable

0L
Long

12.4. Common Property Sets

Given the above representation, it is sufficient to describe a conforming property set representation by specifying the name of the IStorage or IStream that contains the property set (which begins with '\0x05'), the class ID in that IStorage, the format identifier and the list of property identifiers and their allowed types. Optionally, the human-readable names which form the contents of the dictionary may be specified.

12.4.1. Document Summary Information Property Set

The FormatID for the "SummaryInformation" property set is

F29F85E0-4FF9-1068-91AB0-08002B27B3D9.

The stream name should be "SummaryInformation" prepended with the 0x05 (to show that it is a shared property set). Its Property IDs are as follows:

Property Name	**PROP_ID**	**PROP_ID Code**	**Type**
Title	PID_TITLE	0x00000002	VT_LPSTR
Subject	PID_SUBJECT	0x00000003	VT_LPSTR
Author	PID_AUTHOR	0x00000004	VT_LPSTR
Keywords	PID_KEYWORDS	0x00000005	VT_LPSTR
Comments	PID_COMMENTS	0x00000006	VT_LPSTR
Template	PID_TEMPLATE	0x00000007	VT_LPSTR
Last saved by	PID_LASTAUTHOR	0x00000008	VT_LPSTR
Revision number	PID_REVNUMBER	0x00000009	VT_LPSTR
Total editing time	PID_EDITTIME	0x0000000A	VT_FILETIME
Last printed	PID_LASTPRINTED	0x0000000B	VT_FILETIME
Create Time/Date	PID_CREATE_DTM	0x0000000C	VT_FILETIME
Last saved Time/Date[121]	PID_LASTSAVE_DTM	0x0000000D	VT_FILETIME

[121] Some methods of file transfer (e.g. downloading from BBS) do not maintain the file system's version of this information correctly.

Number of Pages	PID_PAGECOUNT	0x0000000E	VT_I4
Number of Words	PID_WORDCOUNT	0x0000000F	VT_I4
Number of Characters	PID_CHARCOUNT	0x00000010	VT_I4
Thumbnail	PID_THUMBNAIL	0x00000011	VT_CF
Name of Creating Application	PID_APPNAME	0x00000012	VT_LPSTR
Security	PID_SECURITY	0x00000013	VT_I4

A property whose value is plain text should use VT_LPSTR, *not* VT_CF with CF representing text. Also note that an application should choose a single clipboard format for a property's value when using VT_CF, which carries only one clipboard format.

Property Guidelines:

- Template refers to an external document containing formatting and styling information. The mechanism by which the template is located is implementation-defined.
- Last Saved By is the Name stored in User Information by the application. For example, suppose Mary creates a document on her machine and gives it to John, then John modifies and saves it; “Mary” is the author, “John” is the last saved by value.
- Revision number is the number of times the **File / Save** command has been performed on this document.
- Create Time / Date is a read-only property: this property should be set when a document is created, but should not subsequently be changed.
- For PID_THUMBNAIL, applications should store data in CF_DIB or CF_METAFILEPICT format.
- By noting the (suggested; that is, application-enforced) security level on the document, an application other than the originator of the document can adjust its user interface to the properties appropriately. An application should not display any of the information about a password protected document, and should not allow modifications to enforced read-only or locked for annotations documents. It should warn the user about read-only recommended if the user attempts to modify properties:

Security Level	**Value**
None	0
Password Protected	1
Read-only recommended	2
Read-only enforced	4
Locked for annotations	8

Appendix A: Registration Database Entries

A.1. Syntax

Keys in the registration database are case insensitive; values are case sensitive (although it matters little for path names and the like). Key names are not localized at all. In fact, very little information here is localized: only the user type names (all forms) and the verb names. Commas are used to separate values into individual items. This convention is not localized (just as win.ini keys and values are not localized).

In the example below, {CLSID} is a shorthand for any class id. In reality the numerical value of the CLSID is put between the {}'s; e.g. {12345678-9ABC-DEF0-C000-000000000046}. All digits are upper case hex and there can be no spaces. This format is the same as the OSF DCE standard and is the result of the StringFromCLSID() routine.

Similarly, {IID} is a shorthand for an interface id. StringFromIID() can be used to produce this string.

Other numbers which appear as keys (usually on the left side of an = sign) are always decimal. Numbers on the right side (values) can be either in decimal or hexadecimal (hex); hex values must be preceded by 0x; e.g., 0xabc123. Negative numbers are always decimal (e.g., -1).

Clipboard formats are represented as a number if the format is built-in to Windows (value < 0xc000) and as a string if not. The string is translated into a number using RegisterClipboardFormat(). (This number can be converted back into the string using GetClipboardFormatName().)

A.2. Programmatic Identifiers

Every OLE2 class that belongs in an Insert Object dialog (hereafter termed an “insertable class”) must have an “programmatic identifier” or ProgID. A ProgID is a string which uniquely identifies a given class. In addition to being used to determine eligibility for the Insert Object dialog, the ProgID can be used as an identifier in a “macro” programming language to identify a class. Finally, the ProgID is also the “classname” used for an OLE2 class when placed in OLE1 containers.

The ProgID string must:

- have no more than 39 characters (i.e.: 39 is OK).
- contain no punctuation (including underscore), with the one exception that it may contain a single period.
- not start with a digit
- be different from the class name of any OLE1 application, *including the OLE1 version of the same application*, if there is one.

CLSIDFromProgID() and ProgIdFromCLSID() can convert back and forth between the two representations. These functions use the registration database do the conversion.

The ProgID must *never* be shown to the user in the user interface. If you need a short human-readable string for an object, call IOleObject::GetUserType(USERCLASSTYPE_SHORT, &szShortName).

See also the “version independent ProgID” at the end of this appendix.

A.3 Insert Object Dialog

Any root key (i.e.: any key immediately under `HKEY_CLASSES_ROOT`) which has either an “**Insertable**” or a “**Protocol\StdFileEditing**” subkey is the ProgID[122] of a class which should be in the Insert Object dialog box. The value of that root key is the human readable name which is displayed in the dialog.

[122] or the OLE1 class name

A.4 Details

When an OLE1 class is inserted for the first time into an OLE2 container, a new subkey "**CLSID**" is added by the OLE2 compatibility layer to the OLE1 registration information. The value given to this key is a CLSID assigned by OLE2 to this OLE1 class.

```
OLE1ClassName = Ole1UserTypeName
    Protocol
        StdFileEditing
            Verb                          // etc...
    CLSID = {CLSID}
```

As described above, OLE2 classes which belong in the Insert Object dialog each have a ProgID. The value of this key is the human readable name which is displayed in the dialog; this should be the same as the MainUserTypeName of the class; see below. If said class is insertable in an OLE2 container,[123] then the ProgID key must have an immediate subkey "**Insertable**"; this key must have no value. If said class is insertable in an OLE1 container, then the ProgID will contain a **Protocol\StdFileEditing** subkey with appropriate further subkeys "**Verb**", "**Server**", etc., as in OLE1. The "**Server**" that should be registered here is simply the full path name to the .EXE of the OLE2 application. An OLE1 container will use this to launch the OLE2 application. The initialization of this application will in turn cause the OLE2 compatibility layer to be loaded. This layer will handle subsequent interactions with the OLE1 container, turning them into OLE2-like requests to the OLE2 application. Thus, in order to be insertable into an OLE1 container, the OLE2 application need take no special action beyond setting up these registration entries.

If the OLE2 application can also handle requests from the Windows 3.1 File Manager, then an appropriate place to put the registration information needed to handle this is under the ProgID.

```
ProgId = MainUserTypeName
    Insertable                          // class is insertable in OLE2 container
    Protocol
        StdFileEditing                  // OLE1 compatibility info, present if and only if objects of this
            Verb                        // class are insertable in OLE1 containers.
                0 = verb 0
                1 = verb 1
            Server = path to .exe
            RequestDataFormats = format,format,format     // NOTE: these formats are strings only[124]
            SetDataFormats = format,format,format         // NOTE: these formats are strings only
    CLSID = {CLSID}                     // The corresponding CLSID. Needed by GetClassFile
    Shell                               // Windows 3.1 File Manager Info
        Print
            ...
        Open
            Command = server.exe %1

.ext = ProgID                           // used by FileManager and by GetClassFile (and thus FileMonikers)
```

Unlike OLE1, the bulk of the OLE2 information is not kept under immediate subkeys of HKEY_-CLASSES_ROOT. Instead, this information is found as subkeys under the "**CLSID**" subkey of HKEY_CLASSES_ROOT. Significantly for product support, this means that the only classes that are explicitly intended to be user-visible appear in the non-verbose mode of the registration database editor REGEDIT.EXE.

These Subkeys of "**CLSID**" are stringized CLSIDs; see StringFromCLSID(). Subkeys of these keys include indications of where the code that services this class is found; see LocalServer, InprocServer, and InprocHandler. A good chunk of the information is used by the default OLE2 handler to return various information about the class when in the loaded state. Examples of this include the Verb, the AuxUserType, and the MiscStatus entries. A couple of subkeys cross reference the information available under the above-described ProgID. In particular, the Insertable subkey should be repeated.

[123] Since OLE2 provides a built-in OLE1<=>OLE2 compatibility layer, it will be rare that an OLE2 class insertable in an OLE2 container will not be insertable in an OLE1 container.

[124] Unlike in OLE2 data formats, where one can also include numbers for the built-in clipboard formats.

```
CLSID
    {CLSID} = Main User Type Name
        LocalServer = path to exe            // local (same machine) server; same as Server = above.
        InprocServer = path to dll           // in process server; relatively rare for insertable classes.
        InprocHandler = path to dll          // in process handler. Use "ole2.dll" to get default OLE2 handler

        Verb                                 // info returned in IOleObject::EnumVerbs(); for example:
            verb number = name, menu flags, verb flags
            -3 = Hide, 0, 1                  // pseudo verb for hiding window; not on menu
            -2 = Open, 0, 1                  // pseudo verb for opening in separate window; not on menu
            -1 = Show, 0, 1                  // pseudo verb for showing in preferred state; not on menu
            0 = &Edit, 0, 2                  // primary verb; often Edit; on menu; possibly dirties object
                                             // MF_STRING | MF_UNCHECKED | MF_ENABLED == 0.
            1 = &Play, 0, 3                  // other verb; on menu; leaves object clean

        AuxUserType                          // auxiliary user types (main user type above)
            form of type = string            // See IOleObject::GetUserType(); for example:
            2 = ShortName                    // key 1 should not be used, for the main user type is found above
            3 = Application name             // contains the human readable name of the application.
                                             // Used when the actual name of the app is needed
                                             // (such as in the Paste Special dialog's result field.
                                             // Example: Acme Draw

        MiscStatus = default                 // def status used for all aspects; see IOleObject::GetMiscStatus()
            aspect = integer                 // exceptions to above; for example:
            4 = 1                            // DVASPECT_ICON = OLEMISC_RECOMPOSEONRESIZE

        DataFormats
            GetSet                           // list of formats for default impl. of EnumFormatEtc;
                                             // very similar to Request/SetDataFormats in OLE1 entries
                n = format ,aspect, medium, flag   // n  is a zero-based integer index;
                                             // format is clipboard format;
                                             // aspect is one or more of DVASPECT_*, -1 for "all";
                                             // medium is one or more of TYMED_*;
                                             // flag is one or more of DATADIR_*.  For example,
                0 = 3, -1, 32, 1             // CF_METAFILE = all aspects, TYMED_MFPICT, DATADIR_GET
                1 = Biff3, 1, 15, 3          // Microsoft Excel's Biff format, version 3, DVASPECT_CONTENT,
                                             //     TYMED_HGLOBAL | TYMED_FILE | TYMED_ISTREAM |
                                             //     TYMED_ISTORAGE, (DATADIR_SET | DATADIR_GET)
                2 = Rich Text Format, 1,1,3  //
            DefaultFile = format             // default main file/object format of objects of this class;
                                             // This is examined in TreatAs scenarios by servers in the
                                             // InitNew case to decide what format to write.

        Insertable                           // when present, the class appears in the Insert Object dialog;
                                             // not present for internal classes like the moniker classes.
        ProgID = ProgID                      // the programmatic identifier for this class. See discussion above.

        TreatAs = {CLSID}                    // see CoGetTreatAs()
        AutoConvertTo = {CLSID}              // see OleGetAutoConvert()

        Conversion                           // Support for Change Type dialog
            Readable
                Main = format,format,format,format, ...
            Readwritable
                Main = format,format,format,format, ...

        DefaultIcon = path to exe, index     // parameters passed to ExtractIcon

        Interfaces = {IID}, {IID}, ...       // Optional. If this key is present, then its values are the totality of the
                                             // interfaces supported by this class: if the IID is not in this list, then
                                             // the interface is never supported by an instance of this class.
```

In order to handle two-way compatibility, the OLE2 compatibility layer as it needs to creates OLE2-style entries for OLE1 classes it discovers. We list this for expository completeness only; no OLE application should ever write these kind of entries directly.

```
CLSID
    {CLSID} = Ole1UserTypeName          // This is an entry auto-generated by OLE2 for an OLE1 class
        Ole1Class = OLE1 class name     // the first time an object of that class is inserted in a 2.0 container,
        ProgId = OLE1 class name        // allowing OLE2 to convert {CLSID} back into a 1.0 class name
```

Finally, a couple of other subkeys of HKEY_CLASSES_ROOT are used. The first of these is FILETYPE, under which are found the entries used by the GetClassFile() function in to pattern match against various file bytes in a file. FILETYPE has {CLSID} subkeys, each of which has a series of further subkeys 0, 1, 2, ... whose values contain a pattern that, if matched, should yield the indicated CLSID.

```
FileType                                // used by GetClassFile()
    {CLSID}
        n = offset, cb, mask, value     // offset and cb are limited to 16 bits. offset can be negative for file end.
                                        // As above, offset and byte count are decimal unless preceded by "0x",
                                        // in which case they are hex. The mask and the pattern are always
                                        // hex, and cannot be preceded by "0x".
                                        // Mask can be omitted, implying a value of all ones. An example:
    {CLSID}
        0 = 0, 4, FFFFFFFF, ABCD1234    // which requires that the first 4 bytes be AB CD 12 34 in that order.
        1 = 0, 4, FFFFFFFF, 9876543     // or requires that they match 9876543.
        2 = -4, 4, , FEFEFEFE           // last four bytes in file must be FEFEFEFE
```

The second subkey of HKEY_CLASSES_ROOT is INTERFACE. Under subkey this are {IID} keys that contain a certain small but important amount of information regarding interfaces.

```
Interface
    {IID} = Textual name of interface   // e.g.: "IOleObject"
        ProxyStubDll = {CLSID}          // Used by OLE2 for interprocess communication.
        NumMethods = integer            // Number of methods in the interface.
        BaseInterface = {IID}           // Interface from which this was derived. Absence of key means IUnknown.
                                        // Key present but empty value means derived from nothing.
```

Some applications (mostly applications support OLE Automation) will also wish to register a "version independent ProgID," a second ProgID referring to their class, but one that does *not* change from version to version but would remain constant over all versions of a given product. This provides a constant name to be used with macro languages which refers to the currently installed version of the application's class.

```
    VersionIndependentProgID = MainUserTypeName
        CLSID = {CLSID}     // the class id of the newest installed version of that class
        CurVer = ProgID     // the ProgID of the newest installed version of that class
```

Index