Microsoft® Software Development Kit (SDK) for 32-bit External Text File Converters[1]

Version 3.0

32-bit Edition for Word 97

Table of Contents

[1] *Microsoft Software Development Kit for 32-bit External Text File Converters* and all of its contents are © Copyright 1996-1997 Microsoft Corporation.
Microsoft, MS, MS-DOS, and Windows are registered trademarks of Microsoft Corporation.

Introduction

The Software Development Kit for External Text File Converters provides the technical information necessary to develop external file converters to support importing and exporting formatted text between Microsoft Word for Windows and foreign binary files. **Microsoft does not offer this kit as part of our product line. Microsoft will not provide technical or any other type of support for this kit.** The kit contains the following components to aid the development process of external text file converters:

1. **Rich Text Format (RTF) Specification** (GC0165.doc)
   The RTF Specification document is the core material needed to write external file converters. This document describes RTF syntax, the semantics of an RTF reader, and the contents of an RTF file, including definitions of RTF keywords. GC0165.doc contains all keywords defined for all versions of Word up to and including Word 97.**Software Development Kit documents** (Sdk.doc and Sdk32.doc) These documents provide specifications and examples of how an external text file converter works with Word and the operating system. This document, Sdk32.doc, details the creation of a 32-bit converter for Win32 and PowerPC for the Macintosh platforms.
2. **Conversions API definitions** (Convapi.h and Converr.h)
   These files define the types, constants and function prototypes for all converter APIs. Use the `#include` command to include the Convapi.h file in your converter sources to be sure your entry points are correct.
3. **Sample sources** (see the Sampcnv and Rtfread directories)
   Refer to the Sampcnv sources for examples of the files that make up a simple text converter. The sample converter sources are intended for use with Microsoft C version 9.0 (Visual C++ 2.0) or later. The Sample32.mak file contains instructions for building the sample converter. The sample converter reads and writes simple text files, is automatically installed into Word, and contains the minimal code needed for a fully functional converter. It implements **InitConverter32, UninitConverter, IsFormatCorrect32, ForeignToRtf32, RtfToForeign32, GetReadNames, GetWriteNames, RegisterApp** and **FRegisterConverter** application program interface (API) functions.

   InitConverter32
   UninitConverter
   IsFormatCorrect32
   ForeignToRtf32
   RtfToForeign32
   GetReadNames
   GetWriteNames
   RegisterApp
   FRegisterConverter

   The Rtfread sources are examples of the files necessary to create a complete RTF reader application. See "Appendix A" of the RTF Specification (GC0165.doc) for more details.

Creating an External Text File Converter

The Importance of RTF

To begin developing a converter, you should familiarize yourself with the RTF specification included with this kit. Text files are imported into Word by converting foreign binary files to RTF. An RTF reader is built into Word to complete the conversion. Word files are exported by converting Word RTF output to foreign binary files. The success of either conversion depends upon accurate generation and interpretation of RTF by the converter. Following these specifications carefully will avoid problems during conversion development and testing.

Writing to Win32

Writing a converter to the Win32 interface will allow a converter to work with Windows 95 and all versions of Windows NT. It is recommended that you limit all your Windows calls to the subset of Win32 common to Windows 95 and Windows NT 3.1. This will allow your converter to operate under all environments for which a 32-bit version of Word exists. You will not be able to take advantage of some Win32 features such as NT security, but you will be able to use features such as memory-mapped file I/O, and full 32-bit memory management.

For more information about porting Win16 converters to the 32-bit Windows API and about the specific differences between the 16-bit and 32-bit conversions interfaces, see "Changes from 16-bit Windows" on page 14 of this document.

Avoid using non-trivial C runtime routines. Most C runtime string functions are safe because they are completely self-contained. More complex runtime components, such as the stdio functions, have initialization requirements that may not be met by a converter DLL. If you must use the C runtime libraries (for example, Windows NT for Intel x86 defines some Windows API calls to use the C runtimes), use the statically-linked C runtimes, not the DLL runtimes. The DLL runtimes may not be properly initialized by the calling application.

Writing to System 7

All PowerPC-based Macintosh computers run System 7 Mac OS or later. It is appropriate to use System 7 features (such as FSSpec file I/O routines) in PowerMac converters. Note that while a version of Word 6.0 for the Macintosh is available for the Power Macintosh, that version only calls 68K converters. Word 97 is the first application to use the interface defined here.

Most of this document is focused on Win32, but the intention is that the System 7 conversion interface be as similar to the Win32 interface as possible. This similarity should simplify maintaining a single code base for the two modern platforms. PowerMac converters are more like Win32 converters than 68K Macintosh converters. It will likely be easier to port a Win32 converter to the PowerMac than to port a 68K Macintosh converter, at least in terms of the application/DLL interface.

Converter Configuration in the Registry

On Win32, information previously stored in .ini files is stored in the registry. In particular, this is true of converter registration information.

Information on converters accessible by any application (that is, shared converters) is stored in subkeys of:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Shared Tools\Text Converters\
```

This information was previously found in WIN.INI. For converters private to a particular application, previously found in an application-specific .INI file, such as WINWORD6.INI, it is stored in subkeys of:

```
HKEY_CURRENT_USER\Software\Microsoft\<appversion>\Text Converters\
```

where `<appversion>` is application specific. In particular, Word 95 used ‘`Word\7.0`’. Word 97 uses ‘`Office\8.0\Word`’.

Under **Text Converters**, information for Foreign-to-RTF converters is found under the subkey ‘`Import`’ and for RTF-to-Foreign converters is found under the **Export** subkey.

Finally, under ‘`Import`’ or ‘`Export`’, as appropriate, is a unique key for each conversion, called its *class name*. A single converter may implement more than one conversion; for example, it may both import and export, and it may import or export multiple subtly different file formats. Hence, more than one class name may be associated with a particular converter. Class names are not seen by the user; therefore, these class names never need to be localized.

Under each class name is the particular registry values for that conversion. The following values (all are strings, or REG_SZ values in registry terms) are required. If a converter implements the **FRegisterConverter** API, it must set these values. If a converter is installed by Word, Word will set the values based on information obtained from the **GetReadTypes** and **GetWriteTypes** API calls to the converter.

Value Name	Value
Name	Description string seen by user in Word’s ‘Confirm Conversions’ dialog. Because this string is seen by users, it should be localized in foreign language versions of a converter.
Path	Absolute path to converter DLL, including the name and extension of the DLL.
Extensions	Space separated list of filename extensions. These extensions specify files that are likely to be of the appropriate binary type for a conversion.

When searching for a converter, an application searches this registry information. An application will likely search through its private converters first, and then it will search the shared converters. A converter may store additional information of its own under any class name key associated with it.

Here is an example set of registry entries for a shared converter that reads and writes WordPerfect 5.0 and 5.1 files. When exporting, it is important for the user to be able to select a specific minor version to save to, and so there are separate entries for 5.0 and 5.1. When importing, the converter can handle files of both minor version types, so only one entry is required.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Shared Tools\
   Text Converters\
       Export\
           WrdPrfctDos50\
               Extensions    “doc wpd”
               Name          “WordPerfect 5.0”
               Path          “C:\Progra~1\…\TextConv\wpft532.cnv”
           WrdPrfctDos51\
               Extensions    “doc”
               Name          “WordPerfect 5.1 for DOS”
               Path          “C:\Progra~1\…\TextConv\wpft532.cnv”
       Import\
           WrdPrfctDos\
               Extensions    “doc”
               Name          “WordPerfect 5.x”
               Path          “C:\Progra~1\…\TextConv\wpft532.cnv”
```

Converter Entry Points

Word converters are implemented as dynamic-link libraries (DLLs). To distinguish converters from other DLLs Word converters end with a .CNV extension, but this is the only difference between a converter and other types of DLLs. Word accesses the converter DLL by using the API functions listed in the following table.

Entry Point	Description	Status
InitConverter32	Initialize the converter.	Required
UninitConverter	Terminate the converter.	Optional
IsFormatCorrect32	Check the format of a file.	Required
ForeignToRtf32	Read binary file or docfile stream and translate to RTF stream.	Required
RtfToForeign32	Read RTF stream and translate to binary file or docfile stream.	Optional
RegisterApp	Parse free-form options from calling application and provide converter’s free-form options to calling	Strongly

	application.	recommended
CchFetchLpszError	Fetch a textual representation of a converter error code.	Optional
GetReadNames	Provide import class names, description strings and file extensions supported by the converter.	Optional
GetWriteNames	Provide export class names, description strings and file extensions supported by the converter.	Optional
FRegisterConverter	Make all appropriate registry entries for this converter, using the same values as provided by **GetReadNames** and **GetWriteNames**.	Optional

These functions must be exported from the converter by name; either with the E**xports** keyword in a .def file or the `__declspec(export)` attribute in the source code. Word will access these routines by name, so it is important to ensure the names are maintained in the resident name table for the converter. This can be assured by not specifying an ordinal value for the function (@1).

For historical reasons, a converter must always provide the import-oriented APIs **IsFormatCorrect32** and **ForeignToRtf32**. If a converter is export-only, provide dummy versions of these APIs. They will not be called, but they must be present for some applications to recognize and load the converter. An import-only converter does not need to provide a dummy **RtfToForeign32** API in order to be recognized.

A converter that does not provide **GetReadNames** or **GetWriteNames** cannot be automatically registered for import or export, respectively.

Argument Types

Argument types and return value types used in the descriptions below are standard Win32 types. **Short** types are 16-bit integers, and **Long** types are 32-bit integers. Strings are understood to be '\0'-terminated. Strings modifiable by the called function, and buffers, are generally passed as handles (as opposed to pointers). This may not always be explicitly stated in the argument description, but can be inferred from the argument type. When modifying such strings, handles must be resized using **GlobalRealloc** to ensure that there is sufficient allocated memory for the data to be returned to Word. If the handle is initially NULL, it cannot be resized and the parameter is not usable.

Each entry point that accepts file names should expect all file name arguments from Word to be in the OEM character set (unless the character set is explicitly negotiated using **RegisterApp**). Because Word configures Windows file APIs to expect their file name arguments to be in the ANSI character set before calling the converter, it is necessary to convert file names from the provided OEM to the ANSI expected by Windows. Call the Win32 **OemToChar** function to do this.

Initializing the Converter

An application will call **InitConverter32** before any other converter entry point, to pass information not conveniently obtainable by the converter any other way, and to permit the converter to perform any necessary global initialization.

long InitConverter32(HWND hwndWord, char *szModule);

<hwndWord>	::=	A handle to the top-level client window of the application to be used as a parent window for the converter. For example, after displaying and closing a dialog box, a converter should reset the focus to this hwnd.
<szModule>	::=	The calling application's module name. The caller must leave the module name accessible throughout the converter session, so that the converter may store this value in a global and access it during a subsequent call. However, it is preferable to copy the string to the converter's data space rather than

relying on it remaining in the caller's space. The module name can be used to modify the converter's behavior for different applications.

This routine should return a non-zero value if the initialization succeeds, and zero if it fails.

Terminating the Converter

An application will call **UninitConverter**, if provided, after a conversion operation is finished, to permit the converter to perform any necessary global finalization.

void UninitConverter(void);

Checking the Format of a Binary File

Word will call the **IsFormatCorrect32** function to ask if the converter recognizes and can convert the contents of a file. Unlike the **Convert** APIs, **IsFormatCorrect32** does not support operating on the contents of an opened docfile stream.

typedef short FCE;

FCE IsFormatCorrect32(HANDLE haszFile, HANDLE haszClass);

<*haszFile*>	::=	A handle to a string containing a fully qualified path to the file to check.
<*haszClass*>	::=	If the file is recognized, the class name of the desired conversion.

The converter should open the file, scan it as necessary to determine if the converter understands the data in the file, and close the file. The function must return 1 (fceTrue) if it recognizes the file, 0 if the file is of an unknown type, or one of the other file conversion error codes described in "Return Codes" on page 13 of this document.

If the converter knows how to convert the file, the converter must fill *haszClass* with the class name for the format that the file should be converted from. This should be one of the class names returned by **GetReadNames**.

Converting a Foreign Binary File or Embedding to RTF

Word converts a foreign binary file to RTF by calling **ForeignToRtf32**.

typedef long (PASCAL *PFN_RTF)(long, long);

short ForeignToRtf32(HANDLE ghszFile, Istorage* pstgForeign, HANDLE ghBuff, HANDLE ghszClass, HANDLE ghszSubset, PFN_RTF lpfnOut);

<*ghszFile*>	::=	A string containing the name of the file to be converted, or NULL if the input is to come from an embedding.
<*pstgForeign*>	::=	A pointer to the IStorage of the embedding being converted, or NULL if the input is to come from a file.
<*ghBuff*>	::=	A handle to the buffer in which the converter should deposit RTF.
<*ghszClass*>	::=	A string which the converter may use to return the write class name for this document to the calling application. The caller may pass NULL if not interested in this information. The write class name is the class name to be used for a subsequent file save.
<*ghszSubset*>	::=	A string containing the desired subset of the file. For example, in a Microsoft Excel worksheet, the subset could be a named range or an area defined by a cell reference. The meaning of this argument is determined by the converter and may be ignored.
<*lpfnOut*>	::=	A pointer to the Word function that is called by the converter whenever it wants to return a chunk of RTF it has generated. The Word function is called by

lRet = (*lpfnOut)(cchBuff, nPercent);

where *cchBuff* is a count of the bytes of RTF data that the converter has placed in *ghBuff*. *nPercent* can range between 0 and 100, representing the estimate made by the converter of how much of the conversion process has been completed. Word will display this estimate in its progress indicator display. Word signals a user cancel request or error by returning a negative value from *lpfnOut.* When this happens, **ForeignToRtf32** should immediately stop converting and return to Word, returning the same negative value. Otherwise, *lpfnOut* returns zero.

For most formats, the calling application must pass a file name to convert. For OLE 2 IStorage-based formats, the calling application may instead pass a pointer to an open IStorage in *pstgForeign*. Exactly one of these two parameters must be used; the other parameter must be NULL. If an IStorage is passed in, the calling application is responsible for calling **Release** on the IStorage after the conversion. The converter will leave the IStorage in the same state it was on entry.

A converter must not assume that Word has verified the file format by calling **IsFormatCorrect32**. When **ForeignToRtf32** is called, the converter should first open the foreign binary file to be converted and decide whether it understands the file's format. If the converter determines that it can convert the input file, the file is converted by passing chunks of RTF to Word through *1pfnOut*. If the conversion is completed successfully, the converter should return 0 . If the converter is unable to complete the conversion, it should return one of the values described in the "Return Codes" section of this document.

If *ghszClass* is non-NULL and if the converter has the ability to write files of the type being imported, it should fill *ghszClass* with the class name for the RTF-to-foreign conversion. This should be one of the class names returned by **GetWriteNames**. Provision of the write class allows the calling application to default to the correct converter if, after importing a document, the user wantsto save the document back out in its original format.

Converting RTF into a Foreign Binary File or Embedding

Word saves documents to non-Word formats by calling **RtfToForeign32**.

typedef long (PASCAL *PFN_RTF)(long, long);

short RtfToForeign32(HANDLE ghszFile, Istorage *pstgForeign, HANDLE ghBuff, HANDLE ghszClass, PFN_RTF lpfnIn);

<*ghszFile*>	::=	A string containing the name of the file to be converted, or NULL if the output is to go to an embedding.
<*pstgForeign*>	::=	A pointer to a destination Istorage, or NULL if the output is to go to a file.
<*ghBuff*>	::=	A handle to the buffer from which the converter should read RTF.

If the calling application is prepared to accept percent complete numbers from the converter on export conversions, it should first register this fact by calling **RegisterApp** with the fRegAppPctComp bit set. It must then pass a handle to a PCVT structure at the start of the buffer to which *ghBuff* is a handle. A PCVT structure has the following form:

```
typedef struct
   {
   short cbpcvt;      // size of this structure
   short wVersion;    // structure version number
   short wPctApp;     // current %-complete
                      // according to application
   short wPctConvtr;  // current %-complete
                      // according to converter
   } PCVT;
```

The application uses the *wPctApp* field in the PCVT structure to send its concept of percent complete (based on percentage of total RTF sent to the converter thus far) to the converter. The converter uses the *wPctConvtr* field in the PCVT structure to send its estimated percent complete number back to the application. The converter will likely base its estimate on the value received from the application.

<*ghszClass*> ::= A string containing the class name for the desired binary file format; provided by a prior call to **ForeignToRtf32** when the document was imported or directly from the registry based on user selection.

<*lpfnIn*> ::= A pointer to the function in the calling application which is called by the converter whenever it needs more RTF to convert. The Word function is called by:

```
lRet = (*lpfnIn)(rgfOptions, 0 /* unused */);
```

rgfOptions is an unsigned array of flags affecting the contents of the RTF stream defined later in this document. The second parameter is unused and should be zero for future compatibility. The return value is usually the number of bytes of RTF text Word has placed in *ghBuff***.** A return value of 0 indicates that there is no more RTF. In this case, the conversion should finish and not request any more RTF. Word signals a user cancel request or error by returning a negative value from *lpfnIn*. When this happens **RtfToForeign32** should immediately stop converting and return to Word, returning the same negative value.

For most formats, the calling application must pass a destination file name to convert into. For OLE 2 IStorage-based formats, the calling application may instead pass a pointer to an open destination IStorage in *pstgForeign*. Exactly one of these two parameters must be used; the other parameter must be NULL.

To convert the input RTF stream, the converter repeatedly calls back to the calling application through *lpfnIn* to obtain RTF text and converts the text into the destination file format. If the conversion completes successfully, the converter should return 0. If the converter is unable to complete the conversion, it should return one of the values described in the "Return Codes" section.

There are some complications when writing to an IStorage that are not relevant in other cases. First, it is possible that the destination IStorage is not initially empty. Therefore, **RtfToForeign32** should first **EnumElements** on the IStorage it receives and call **DestroyElement** on each of these elements to empty the IStorage before conversion begins.

Also, the IStorage passed in by the application must be opened in transacted mode (STGM_TRANSACTED), and must be opened read-write (STGM_READWRITE). It must be unmodified since the last **Open** or **Commit** at the time it is passed to **RtfToForeign32**. If the **RtfToForeign32** call returns failure, the application must call **Revert** on the IStorage in order to restore the IStorage to its state before the failed conversion. If **RtfToForeign32** returns success, the host must call **Commit** on the IStorage to save the new, converted document. The converter itself should never call **Release**, **Commit**, or **Revert** on the IStorage at *pstgForeign*. It should, however, commit changes to each substorage and stream before calling the **Release** function; otherwise, those changes cannot be committed by the calling application.

Bit	Name	Description
0	fPicture	Word will use binary encoding for pictures if fPicture is set. If fPicture is clear, the converter is assumed not to be able to convert graphic objects, and Word removes them from the RTF stream. This implies that if a converter handles pictures at all, it must be prepared to handle them encoded as binary.
1	fLayoutInfo	Word will provide layout information in the RTF stream if this bit is set. Layout information consists primarily of RTF soft

		line-break control words.
2	fPctComplete	The converter will provide percent complete numbers if this bit is set. The converter should set this bit only if the application called **RegisterApp** with the fRegAppPctComp bit set and a PCVT structure is available.
3-31	unused	Available for future use. Must be set to zero.

Negotiating Converter/Application Preferences

In theory a converter should behave identically in all situations, providing the richest conversion possible for each document. RTF is designed to allow multiple representations of a single item, and an application can skip data it is not interested in or doesn't understand. In practice, however, there are reasons for wanting the converter to behave differently at times. For example, when importing, it can be more efficient to omit data from the generated RTF that the calling application can't or won't use. Also, because the conversions API has been extended over time, older applications may not support specific features in the current API, in which case the converter must not rely on or use these features.

Caller Identification

A converter private to a particular application will most likely be specifically tailored to work with that application. A shared converter, however, may want to modify its behavior based on which application is calling it. The identity of the calling application can only be deduced from the *szModule* parameter passed to **InitConverter32**. The application module name is not available through any Win32 function. However, Win32 APIs can be used to get version information about the calling application if the application has the Windows version resource.

To get version information on Win32, use **GetCurrentProcess** to get a handle to the current process. Use this handle with **GetModuleFileName** to get a full path and file name to the calling application. Finally, use this path information with **GetFileVersionInfoSize** and **GetFileVersionInfo** to obtain specific version information.

Similarly, the application may want to get version information from the converter. Because the application already knows the file name of the converter, it can use **GetFileVersionInfoSize** and **GetFileVersionInfo** directly. The application can then call **VerQueryValue** with *InternalName* to get the module name and gather other version information about the converter.

Negotiating Preferences

The **RegisterApp** API is the means by which the converter and application can negotiate details about the conversion to follow. The application calls the converter with a list of preferences from the application and the converter returns a list of its own preferences. Having both sets of preferences, both the application and the converter can deduce the correct behavior for this particular conversion. The lists are self-describing and may be extended in future. This mechanism is designed to be used for all new refinements to the conversions API, and there should consequently be no extensions to older mechanisms such as the flags in PFN_RTF or even the flags to RegisterApp. It is strongly recommended that all converters implement this API. However, not all legacy applications will call it.

HGLOBAL RegisterApp(long lFlags, void *lpRegApp);

<lFlags>	::=	A group of flags specifying options to the converter, defined below.
<lpRegApp>	::=	A pointer to a list of application preferences. May be NULL, indicating there are no application preferences.

The converter will return an HGLOBAL handle to a **GlobalAlloc**-retrieved list of its own preferences. The converter can return NULL if it has no preferences. The calling application must use **GlobalFree** on this list of converter preferences if non-NULL to avoid memory leaks. The converter should not free the list of application preferences; the application is responsible for that memory also.

Both preference lists have similar structure. A preference list is a packed string of bytes, with no padding. The first two bytes are a short value containing the length of the entire string, including the size of the size short itself. The rest of the string consists of arbitrary records. Each record begins with a size byte, followed by an opcode byte. Each size byte includes itself in the size. The rest of each record contains record-specific data. See the structure below for an example.

To parse the preferences list of the application, a converter must look at the list byte-by-byte, not exceeding the number of bytes in the size word, and it must neither assume any particular ordering of opcodes, nor should it assume the presence of any particular opcodes. Because each record includes its own size, a converter can and should skip unknown opcodes. Parsing the preferences list is somewhat analogous to parsing RTF in this respect. The application must parse the preference list from the converter in a similar way.

Because the opcode list is transient and is communicated between two components running on the same system, there is no need for byte-swapping. All values should be in native byte order.

While parsing of preference lists must be done byte-by-byte, the lists do not need to be constructed that way. Typically, the list generated by a converter is fairly static. It is convenient to build it using a structure:

```
#pragma pack(1)
typedef struct
      {
      short cbStruct;         // size of entire structure

      // version of Word with which converter's Rtf is compliant
      char cbSizeVer;         // == 1 + 1 + 2 + 2
      char opcodeVer;
      short verMajor;         // major version of Word
      short verMinor;         // minor version of Word

      // character set we want all file names to be in
      char cbSizeCharset;     // == 1 + 1 + 1
      char opcodeCharset;
      char charset;

      // additional values can be added here
      } REGAPPRET;
#pragma pack()
```

This mechanism allows the set of opcodes to grow in a backwards-compatible fashion. See the conversions API file Convapi.h for the complete list of currently defined opcodes and their record structures. These are the records defined as of Word 97. Many are esoteric and not of general interest. The ‘small’ valued opcodes are for the converter to pass to Word. The opcodes valued 0x80 and above are for Word to pass to the converter.

Byte	Name	Description
0x1	Ver	Specifies major and minor version of Word with which the RTF of the converter is compliant. A converter using this opcode must specify at least Word 6.0 level RTF; newer versions of Word are not guaranteed to be able to provide RTF compatible with older readers. This opcode takes two short arguments, the major and minor version numbers.
0x2	Docfile	Specifies whether or not the converter can handle Word doc ument files, and whether or not the converter can handle regular files. This opcode takes one short argument. Bit 0 indicates whether the converter accepts docfiles, bit 1 indicates whether the converter accepts non-docfiles.

Byte	Name	Description
0x3	Charset	Specifies the character set (either ANSI or OEM) in which file names should be encoded. The default is OEM, in which case the converter must convert the OEM file names to ANSI itself, using the Win32 **OemToChar** function. Using this opcode to specify that the converter can handle ANSI file namesis preferable, as the conversion from ANSI to OEM and then back to ANSI is flawed, making some file names impossible to pass to a converter. This opcode takes a single byte argument which is the character set.
0x4	ReloadOnSave	The presence of this opcode specifies that Word should reload the document from the newly-saved file after an export conversion. This opcode takes no arguments.
0x5	PicPlacehold	The presence of this opcode specifies that Word should include placeholder pictures with size information when emitting RTF for INCLUDEPICTURE fields. This opcode takes no arguments.
0x6	FavourUnicode	The presence of this opcode specifies that Word should prefer Unicode over other character representations, such as DBCS, when emitting RTF. This opcode takes no arguments.
0x7	NoClassifyChars	The presence of this opcode specifies that Word should not break text runs by character set classification. This opcode takes no arguments.
0x80	File name	Word exports documents through converters to temporary files and renames the temporary files after the conversion completes successfully. This opcode takes one variable length argument which specifies the final filename to which Word will rename the finished conversion. The filename is not '\0'-terminated. Its length is inferred from the opcode's cbSize field. This filename is always in the ANSI character set.
0x81	InterimPath	The presence of this opcode indicates that Word will move the file after the export operation completes; for example to an FTP server. This opcode takes no arguments.

RegisterApp still supports the flags passed from the application to the converter in its first argument. This is why the function was originally provided.

Bit	Name	Description
0	fRegAppPctComp	The application sets this flag if it is prepared to accept percent complete numbers from the converter on RtfToForeign32 calls. When this bit is set, the application is required to pass a valid handle to a PCVT structure inside the *ghBuff* buffer on the **RtfToForeign32** call. The setting of this bit does not require that the converter provide

Bit	Name	Description
		percent complete numbers but it can if you like.
1	fRegAppNoBinary	This flag is set if the application is not prepared to deal with binary data in the RTF stream from the converter. If this flag is set, the converter will provide all picture data in hexadecimal form rather than binary. Because some older RTF readers do not correctly handle binary data, this flag is assumed set if **RegisterApp** is not called.
2	fRegAppPreview	This flag is set if the converter is being called in a preview mode (such as the Find preview in Word). The converter can respond to this setting by not displaying dialog boxes, using default responses, and taking other actions to enhance performance when previewing a document.
3	fRegAppSupportNonOem	If this flag is set, the application is prepared to provide non-OEM (that is, ANSI) file names. If this flag is set by the application *and* the converter provides the Charset opcode with an ANSI argument in its preferences list, then it is understood by both the application and the converter that file names will be passed in ANSI.
4	fRegAppIndexing	This flag is set to indicate that the calling application is only indexing textual content, and does not need layout and other information. When this flag is set, the converter can omit most RTF to significantly speed up the conversion.
5-31	unused	Available for future use. Must be set to 0.

Installing a Converter Without User Interaction

Usually, converters are installed by the Setup program of an application. The Setup program ensures that the converter components are copied to a suitable location and that the appropriate registry entries are made. However, a user shouldn't have to run a Setup program to install a new converter. Word will automatically install new converters found in its directory or the shared converter directory (usually C:\Program Files\Common Files\Microsoft Shared\TextConv) the first time it attempts to convert a file in a given session. If a user copies a converter DLL to one of the appropriate directories, all that remains is to make registry entries for it.

If a converter supplies the following APIs, Word (and other applications) can make the registry entries on the behalf of the converter. This is known as *auto-installing* the converter. **GetReadNames** returns Import subkey registry entry information and **GetWriteNames** returns Export subkey registry entry information.

void GetReadNames(HANDLE haszClass, HANDLE haszDescrip, HANDLE haszExt);

void GetWriteNames(HANDLE haszClass, HANDLE haszDescrip, HANDLE haszExt);

<*haszClass*>	::=	A string in which the converter must return a list of class names recognized by this converter.
<*haszDescrip*>	::=	A string in which the converter must return a list of description strings, paralleling the classes in *haszClass*. The description strings are displayed to the user, and must be localized.
<*haszExt*>	::=	A string in which the converter must return a list of file name extensions, paralleling the classes in *haszClass.*

The lists in each argument are sequences of '\0'-terminated strings, the last of which is empty; that is, each list ends with two '\0's. The same number of strings must be placed in each of these buffers even if this requires duplicating some of the contents. In **GetReadNames**, multiple extensions may be valid for a single class name. The various extensions within a single string are separated by spaces. It is not valid to provide multiple extensions for a single class name in **GetWriteNames**.

The buffers passed to **GetReadNames** and **GetWriteNames** are initialized by the caller with a single empty string in each. If the converter cannot reallocate the buffers, it should return to Word. If any buffer has an empty string on return, the call is considered to have failed and no entry is made in the registry.

Although read classes and write classes are conceptually distinct, some applications (notably Word for Windows 6.x) fail to distinguish between them. For this reason, converters should be careful not to use a single string both as a read class and as a write class. The complete set of read and write class names should contain no duplicates.

As an example, the WordPerfect converter discussed in the "Converter Configuration in the Registry" section of this document would return the information listed in the following table.

	GetReadTypes	**GetWriteTypes**
haszClass	"WrdPrfctDos\0\0"	"WrdPrfctDos50\0WrdPrfctDos51\0\0
haszDescrip	"WordPerfect 5.x\0\0"	"WordPerfect 5.0\0WordPerfect 5.1 for DOS\0\0"
haszExt	"doc\0\0"	"doc wpd\0doc\0\0"

Auto-installing using **GetReadNames** and **GetWriteNames** requires effort by the application on behalf of the converter. This simplifies the converter but requires applications to have 'too much' knowledge about 'converter stuff.' The new preferred mechanism for registering a converter is the **FRegisterConverter** API.

long FRegisterConverter(HANDLE hkeyRoot);

<*hkeyRoot*>	::=	A handle to a registry key, under which the converter should register itself. This key is at the level of 'Shared Tools' or '<appversion>' in the "Converter Configuration in the Registry" example on page 3 of this document.

When called by an application, **FRegisterConverter** will attempt to make all the appropriate Import and Export registry entries for this converter. The values registered should be the same as those returned by **GetReadNames** and GetWriteNames. The function must return 1 (fceTrue) if it was able to update the registry, otherwise 0 . See the implementation of **FRegisterConverter** included in the sample converter sources for specific details concerning accessing the registry with Win32 APIs.

Return Codes

The following table describes the error codes that may be returned by **IsFormatCorrect32**, **ForeignToRtf32**, and **RtfToForeign32**. These errors may be generated by the converter, or returned to the converter by a Word callback, prompting the converter to shut down and, in turn, re-return the error to Word. A converter should not post its own error dialog boxes. When an error occurs, the converter should silently return a code to the calling application, which is then responsible for displaying an appropriate message. This insures that conversion errors are displayed in a manner consistent with the user interface of the calling application.

Code	Name	Description
1	fceTrue	**IsFormatCorrect32** recognized the input file.
0	fceNoErr	**IsFormatCorrect32** did not recognize the input file. Operation completed successfully for other APIs.
-1	fceOpenInFileErr	**IsFormatCorrect32** or **ForeignToRtf32** was unable to open the input file.

-2	fceReadErr	There was an error reading from a file.
-4	fceWriteErr	There was an error writing to a file.
-5	fceInvalidFile	There was invalid data in a document being imported through **ForeignToRtf32**.
-8	fceNoMemory	The converter was unable to allocate memory for an operation.
-12	fceOpenOutFileErr	**RtfToForeign32** was unable to create its output file.
-13	fceUserCancel	The user cancelled the conversion operation.
-14	fceWrongFileType	**ForeignToRtf32** did not recognize the input file while attempting to convert it.

The error codes -3, -6, -7, -9, -10, and -11 are obsolete, and should not be returned.

For historical reasons, an application must be aware of these particular error codes and their meanings, and be able to inform the user about the error without additional information from the converter. An application may want to map any obsolete errors that are returned as indicated in Converr.h.

A converter can define and return additional error codes of its own, with numeric values more negative than -14. An application cannot be expected to know the meaning of any such codes, and should report a generic "Cannot open file" message. However, if the converter supplies the following optional API, the calling application may choose to call it to obtain a textual representation of any return value and use this to provide better diagnostics to the user.

long CchFetchLpszError(long fce, char *lpszError, long cb);

<fce>	::=	An error code previously returned by the converter.
<lspzError>	::=	A string into which the textual representation of *fce* should be copied.
<cb>	::=	The size of *lpszError* in bytes.

If a converter chooses to provide this function, it must provide textual representations for all of the error values it returns, including the standard codes defined above. If *lpszError* is not large enough to contain the textual representation (including a terminating '\0'), or if *fce* does not refer to a valid error code, the converter should return 0. Otherwise, the converter should return the length of the error text in bytes, not counting the terminating '\0'. Because the strings provided by **CchFetchLpszError** may be displayed to the user, they need to be localized for foreign language versions of the converter.

Tips

Changes from 16-bit Windows

Writing DLLs for 32-bit Windows is significantly different from writing for 16-bit Windows. Here are some things to keep in mind when porting a converter from 16-bit.and also when writing a 32-bit converter from scratch.

- The CoRoutine Manager is no longer used. Win32 allows an application to allocate a large stack, so there is no need for a converter to create its own stack. Not using the CoRoutine manager simplifies converter entry points, as well as eliminating references to LCallOtherStack when calling Windows and application callbacks.
- 32-bit converters provide a different set of entry points. Each of the principal 16-bit entry points has a 32-bit counterpart. Most of these are obviously named. A function was renamed if any changes were made to its argument list or semantics. There are also numerous new entry points that have no 16-bit equivalent. Pay particular attention to **GetReadNames** and **GetWriteNames** (which replace **GetIniEntry** and **GetClassNames**) because these have changed more significantly than most functions.

- All converters should implement **RegisterApp**, and all converter-calling applications should call **RegisterApp**. This is a change from Win16, where RegisterApp was optional on both the converter and application sides.
- A Win16 converter reports an error to the user through a dialog box managed by the converter before it shuts down. Win32 converters should only return an appropriate error code to the calling application. The application can then handle reporting the error to the user. The application may call **CchFetchLpszError** to obtain suitable error message text from the converter.
- The set of error codes a converter can return has changed. Several redundant or ill-defined codes have been eliminated and some new codes have been defined. A 32-bit converter should not return any of the obsolete codes that have been eliminated.
- Configuration and other information previously kept in .ini files is now stored in the registry. The Win32 APIs for manipulating the registry are significantly different from the APIs used to handle .ini files.

The following items are not specific to writing converters, but are relevant to any Win16-to-Win32 porting effort:

- Win32 does not have memory models the Win16 does. All applications run in a flat 32-bit address space. This is one of the major benefits of Win32, but code that depends on specifics of memory models and the segmented Intel x86 architecture will have to be modified.
- Many calls to Windows APIs will have to be modified, because Win32 tends to have 'wider' parameters than Win16.
- Win16 DLLs have **LibMain** and **WEP** functions, which are replaced by **DllMain** on Win32. DllMain must be explicitly exported (through a .def file, for example) in order to get called when the DLL is loaded or unloaded. If a DLL is not fully re-entrant (typically the case), DllMain should enforce that there is only a single active caller. See the included sample converter sources for a DllMain that handles this. Note also that the .def file must specify a SINGLE data segment for the multiple-caller detection to function correctly.
- Win32 is case-sensitive with respect to exported DLL function names; Win16 is not, despite documentation to the contrary. In Win32, arguments to GetProcAddress must have the same case as function names exported in the .def file of a DLL.

Rebasing

When loaded by an application, a DLL is loaded at some address in memory. Traditionally, the operating system picks an appropriate address (that is, the start of some currently unused region) at which to load the DLL and then fixes up addresses within the freshly loaded DLL image to reflect the location of that instance.

On Win32 systems, each DLL has a preferred address at which to load. If a DLL can be loaded at its preferred base address, no fixups need to be made to the image because all the memory references within the DLL image are initially correct. Not needing to fix up a DLL can dramatically improve its load time and subsequent paging performance.

To maximize the chance that all DLLs in an application can load at their preferred addresses, it is important to consider which DLLs can be active simultaneously and assign them non-overlapping ranges of memory. Once assigned a range, it is simple to specify to the linker at build time for a particular DLL that the start address of the address range of the DLL should be used as its preferred base address.

The address range allocated for Word converters is **0x01400000 to 0x01800000**. Word never has two converters loaded at the same time, so in general, converters can be based at the start of the allocated range, 0x01400000. Some converters may themselves load other DLLs as part of their implementation. In this case, the various DLLs should be based so that they do not overlap. Allowing a gap of 0x00200000 between DLLs should be sufficient. Thus, it is recommended that any second DLL be based at 0x01600000.

To specify the base address when building a converter with Microsoft tools, use the **/base** switch of the linker. For example, for most filters, add the text **/base:0x01400000** to the linker command line.