Ship Manager API

Tagged Transaction Parser Developer's Guide

Table of Contents

FedEx Ship Manager API

Tagged Transaction Set Parser

Introduction

The Tagged Transaction Set Parser can be used to build Tagged Transaction Set (TTS) requests for passing to the FedEx Ship Manager API, and to parse the TTS responses returned from the API. It eliminates the need for the client program developer to be aware of the specific syntax rules of the TTS language. Standard DLL and COM interfaces are provided for Windows. Both reside in the same DLL, FDXParser.dll. The Solaris UNIX and LINUX interfaces reside in libctsparser.so and libctsparser.a.

The Standard Library Interface

Functions

OS Compatibility: Windows, Solaris UNIX, and LINUX.

Parser Creation and Destruction

void* WINAPI createParser();

Creates an instance of the parser. Returns a pointer to that instance that must be passed into the other functions to work on that instance.

void* WINAPI copyParser(void *parser);

Creates a new instance of the parser with state equal to that of the given one.

void WINAPI destroyParser(void *parser);

Destroys the given parser. Must be called for every parser created, whether with createParser() or copyParser().

Building/Parsing

void WINAPI build(void *parser, char *transBuffer, long bufferSize, long *transLength);

Constructs and returns a TTS request containing the fields (tag/value pairs) that have been set with the set functions. Places up to bufferSize bytes of the constructed request in the buffer at transBuffer. Places the total length of the request into the long variable at transLength. If that total length exceeds bufferSize, then the buffer was not large enough.

bool WINAPI parse(void *parser, const char *transaction, long *errorOffset);

Parses characters in the buffer at transaction until a syntax error has been detected or a null character has been encountered. If a syntax error is found, returns false and places the offset into the buffer at which the error was found into the long variable at transLength. Otherwise, returns true. Note that the bool return type is a one-byte integral type.

void WINAPI clear(void *parser);

Removes all fields, whether gotten from set calls or from parsing a response.

Accessing Transaction Type

void WINAPI setTransactionType(void *parser, const char *type);

Sets the TTS transaction type to the null-terminated string pointed to by the second argument. Note that TTS transaction types are NOT integers. All transaction types defined as of this writing happen to contain only the characters ‘0’ through ‘9’ (some with leading ‘0’s), but there is no guarantee that that pattern will be maintained. The transaction type must be set with this function. It cannot be set by setting tag “0”.

void WINAPI getTransactionType(void *parser, char *typeBuffer, long bufferSize, long *typeLength);

Gets the currently set TTS transaction type. If called after parsing a response, that will be the transaction type of the response. Places up to bufferSize bytes of the transaction type in the buffer at typeBuffer. Places the total length of the transaction type into the long variable at typeLength. If that total length exceeds bufferSize, then the buffer was not large enough. If the transaction type is not set, places a single null character in the buffer and sets the variable at typeLength to zero.

Setting and Getting Text Values

void WINAPI setField(void *parser, const char *tag, const char *value);

Associates the given null-terminated text value with the given null-terminated tag.

void WINAPI setFieldInstance(void *parser, const char *tag, const char *value, short index);

Associates the given null-terminated text value with the instance of the given null-terminated multi-occurrence tag having the specified zero-based index. One would set the third instance, for example, by passing 2 as the index.

void WINAPI getField(void *parser, const char *tag, char *valueBuffer, long bufferSize, long *valueLength);

Gets up to bufferSize characters of the text value associated with the given tag into valueBuffer, and sets the variable at valueLength to the total length of the value. If that length exceeds bufferSize, then the buffer was too small.

void WINAPI getFieldInstance(void *parser, const char *tag, char *valueBuffer, long bufferSize, long *valueLength, short instance);

As getField, but gets the instance having the specified zero-based index.

Setting and Getting Binary Values

void WINAPI setBinaryField(void *parser, const char *tag, const char *value, long valueLength);

Associates the number of bytes specified in valueLength at the address given in value with the given null-terminated tag. This function must be used with tags that require binary data, including “188”, “411”, “1005”, “1367” and “1471”.

void WINAPI setBinaryFieldInstance(void *parser, const char *tag, const char *value, long valueLength, short instance);

As setBinaryField, but sets the instance having the specified zero-based index.

void WINAPI getBinaryField(void *parser, const char *tag, char *valueBuffer, long bufferSize, long *valueLength);

Gets up to bufferSize bytes of the stream of bytes associated with the given tag into valueBuffer, and sets the variable at valueLength to the total length of the value. If that length exceeds bufferSize, then the buffer was too small. Must be used with tags that require binary data.

void WINAPI getBinaryFieldInstance(void *parser, const char *tag, char *valueBuffer, long bufferSize, long *valueLength, short instance);

As getBinaryField, but gets the instance having the specified zero-based index.

Checking for Existence of Fields

bool WINAPI fieldExists(void *parser, const char *tag);

Returns true if and only if the given field (tag/value pair) exists. Note that the bool return type is a one-byte integral type.

bool WINAPI fieldInstanceExists(void *parser, const char *tag, short instance);

Returns true if and only if the instance of the given field having the specified zero-based index exists. Note that the bool return type is a one-byte integral type.

short WINAPI getInstanceCount(void *parser, const char *tag);

Returns the number of instances of the given field.

Enumerating All Tags

void WINAPI enumerateTags(void *parser);

Initializes a mechanism that will allow a sequence of calls to getNextTag() to be made to retrieve all of the tags that have been set (or parsed from a response).

bool WINAPI getNextTag(void *parser, char *tagBuffer, long bufferSize, long *tagLength);

Gets the next tag from the enumeration created by enumerateTags(). Returns false when there are no more tags to be gotten. Places no more than bufferSize characters of the tag into the buffer at tagBuffer. If the tag length placed into the variable at tagLength exceeds bufferSize, then the buffer was not large enough. Note that the bool return type is a one-byte integral type.

Usage

Typically, you would first call createParser(), which returns a handle that must be passed into the other functions. You then call setTransactionType(), passing the transaction type code of the transaction that you wish to construct, such as “021” for a shipping transaction (for those familiar with Tagged Transaction Set syntax: the transaction type is NOT set by setting tag “0”, nor should tag “99” ever be set). Then you call setField() to set the value for each TTS tag. You can find the tags appropriate for each type of transaction, and the valid values for them, in the most recent FedEx Tagged Transaction Guide. When you have set all the desired tags, call build() to create a request to send to FedEx via the FedEx Ship Manager API. After you receive a response from the API, pass it into the parse() function. Now you can call the get functions as needed to retrieve the values for the tags that you are expecting to find in the response. Finally, you need to destroy the parser by passing its handle to destroyParser().

You can create multiple instances of the parser and use them concurrently. Because the parser is threadsafe, you can even use a single instance of it within multiple threads concurrently.

The COM Interface

Names and IDs

Class Name – CTSParser

ProgID – Parser.CTSParser.1

VersionIndependentProgID – Parser.CTSParser

Interface Name – ICTSParser

Functions

OS Compatibility: Windows ONLY.

Building/Parsing

HRESULT Build([out,retval] BSTR *transaction);

Constructs and returns a TTS request containing the fields that have been set with the set functions.

HRESULT: S_OK

HRESULT Parse([in] BSTR transaction, [out, retval] long *errorOffset);

Parses the Unicode string in transaction, making values available for getting. If a syntax error is found, returns the offset at which the error occurred. If succeeds, returns offset of last character in transaction plus 1.

HRESULT: E_FAIL if syntax error found, S_OK otherwise

HRESULT Clear();

Removes all fields, whether gotten from set calls or from parsing a response.

HRESULT: S_OK

Accessing Transaction Type

HRESULT SetTransactionType([in] BSTR transactionType);

Sets the TTS transaction. Note that TTS transaction types are NOT integers. All transaction types defined as of this writing happen to contain only the characters ‘0’ through ‘9’ (some with leading ‘0’s), but there is no guarantee that that pattern will be maintained. The transaction type must be set with this function. It cannot be set by setting tag “0”.

HRESULT: S_OK

HRESULT GetTransactionType([out,retval] BSTR *transactionType);

Gets the currently set TTS transaction type. If called after parsing a response, that will be the transaction type of the response. If the transaction type is not set, returns an empty BSTR.

HRESULT: S_OK

Setting and Getting Text Values

HRESULT SetField([in] BSTR tag, [in] BSTR value);

Associates the given text value with the given tag.

HRESULT: S_OK

HRESULT SetFieldInstance([in] BSTR tag, [in] short index, [in] BSTR value);

Associates the given text value with the instance of the given multi-occurrence tag having the specified zero-based index. One would set the third instance, for example, by passing 2 as the index.

HRESULT: S_OK

HRESULT GetField([in] BSTR tag, [out,retval] BSTR *value);

Gets the text value associated with the given tag into the BSTR at value.

HRESULT: S_OK

HRESULT GetFieldInstance([in] BSTR tag, [in] short index, [out,retval] BSTR *value);

Gets the text value associated with the instance of the given tag having the specified zero-based index into the BSTR at value.

HRESULT: S_OK

Setting and Getting BinaryValues

HRESULT SetBinaryField([in] BSTR tag, [in] BSTR value);

Associates the given binary data value with the given tag.

HRESULT: S_OK

HRESULT SetBinaryFieldInstance([in] BSTR tag, [in] short index, [in] BSTR value);

Associates the given binary data value with the instance of the given multi-occurrence tag having the specified zero-based index. One would set the third instance, for example, by passing 2 as the index.

HRESULT: S_OK

HRESULT GetBinaryField([in] BSTR tag, [out,retval] BSTR *value);

Gets the binary data value associated with the given tag into the BSTR at value.

HRESULT: E_POINTER if pointer value is NULL, otherwise S_OK

HRESULT GetBinaryFieldInstance([in] BSTR tag, [in] short index, [out,retval] BSTR *value);

Gets the binary data value associated with the instance of the given tag having the specified zero-based index into the BSTR at value.

HRESULT: E_POINTER if pointer value is NULL, otherwise S_OK

Checking for Existence of Fields

HRESULT FieldExists([in] BSTR tag, [out,retval] VARIANT_BOOL *result);

Returns result of true if and only if the given field (tag/value pair) exists (has been set with a set function or has been parsed out of a response with parse()).

HRESULT: S_OK

HRESULT FieldInstanceExists([in] BSTR tag, [in] short index, [out,retval] VARIANT_BOOL *result);

Returns result of true if and only if the instance of the given field having the specified zero-based index exists.

HRESULT: S_OK

HRESULT GetInstanceCount([in] BSTR tag, [out,retval] short *count);

Returns the number of existing instances of the given field.

HRESULT: S_OK

Enumerating All Tags

HRESULT EnumerateTags();

Initializes a mechanism that will allow a sequence of calls to getNextTag() to be made to retrieve all of the tags that have been set (or parsed from a response).

HRESULT: S_OK

HRESULT GetNextTag([out, retval] BSTR *tag);

Gets the next tag from the enumeration created by enumerateTags(). Returns an empty BSTR when there are no more tags to be gotten.
HRESULT: S_FALSE if no tags remain to be gotten, S_OK otherwise

Usage

The usage of the COM interface is functionally equivalent to that of the standard DLL. However, there are of course no creation/destruction functions, and no parser handle to be passed into the other functions. An instance of the COM class is created and its methods called.

Function names are capitalized.

Text strings are Unicode, passed as BSTRs. However, the BSTRs passed into and out of the binary versions of the set and get functions must NOT contain Unicode text strings or be treated as such. They contain raw binary data. Any binary data that you wish to pass into one of these functions must NOT be converted by the process whereby each ASCII character (byte) becomes a two-byte Unicode character, but rather, loaded directly into a BSTR using whatever facility the language being used offers for that purpose.

The COM interface supports extended error information.