Interactive Financial Exchange

Version 1.0.1

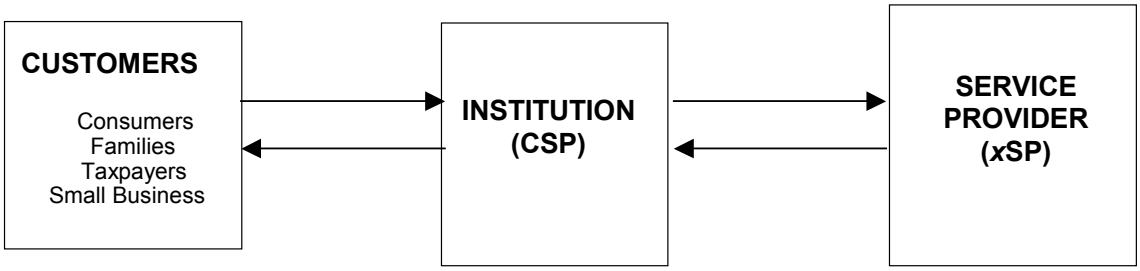
XML Implementation Specification
April 26, 2000

Contents

1 Overview

1.1 Introduction

The Interactive Financial Exchange (IFX) Specification provides a robust and scalable framework for the exchange of financial data and instructions independent of a particular network technology or computing platform. The information-sharing potential of IFX has been designed to support communication not only between a Financial Institution and its customers, but also between a Financial Institution and its Service Providers.



IFX is an open specification that anyone can implement: any Service Provider, software developer, or other party. It uses widely accepted open standards for data formatting (such as XML), connectivity (such as TCP/IP and HTTP), and security (such as SSL).

The IFX Business Message Specification defines the request and response messages used by each financial service as well as the common framework and infrastructure to support the communication of those messages. This specification does not describe any specific product implementation.

This XML Implementation Specification is a companion document to the IFX Business Messages Specification 1.0.1. It defines the specific XML conventions that govern the syntax specified in the accompanying Document Tag Definition (DTD).

1.1.1 Governing Principles

We applied several principles in the development of the XML implementation of IFX:

- **Faithfully render the business messages –** Unless a compelling rationale dictates otherwise, the XML implementation should use the same message, aggregate, and tag structure as the business message specification expresses. One departure from the aggregate and tag structure is in the representation of time-related datatypes. These aggregates have been defined to be consistent with the XML schema definition of the timeInstant datatype, which in turn references the ISO 8601 standard representation of time.
- **Favor the format of the business messages** over XML optimizations such as use of attributes. We recognize the potential performance trade-off here and may revisit this trade-off in future releases.
- **Enhance extensibility through XML namespaces –** XML namespaces complement the extensibility mechanism defined in the business messages specification.
- **Rely on channel level encryption (such as SSL or SMIME) for privacy and data integrity –** IFX provides built-in mechanisms for authentication, but does not provide facilities to protect privacy and guarantee data integrity between end-points. The XML implementation relies on channel level mechanisms for these aspects of message security.
- **Support batch and interactive styles of communication –** While the IFX business messages manifest themselves as request / response pairs, applications may choose to issue one XML IFX request at a time and then wait for the associated response; or applications may transmit numerous requests at a time and likewise receive and process a batch of response messages. We have intentionally minimized the XML overhead to accommodate both forms of interaction.
- **Remain application protocol independent –** The XML implementation of IFX is independent of the protocols used to transport the messages between the client and server computers. While IFX XML will probably use HTTP in most situations, FTP or SMTP, or any number of queuing systems could be used as message transport protocols as well. Likewise, while IFX / HTTP(S) / TCP / IP will probably be the most common stack used for communication of IFX messages, other stacks may be used as well (for example, IFX / MQ / SNA).

The key rule of IFX syntax is that each tag is either an element or an aggregate. Data is contained between the element start tag and its respective end tag. An aggregate tag begins a sequence of enclosed elements or inner aggregates, which must end with a matching tag; for example, <Aggregate> ... </Aggregate>.

The file sent by IFX does not require any white space between adjacent tags. White space anywhere within an element is significant.

1.2 IFX XML Data Transport over IP

The design of IFX is as a client and server system. An end-user uses a client application to communicate with a server at a Customer Service Provider. The form of communication is requests from the client to the server and responses from the server back to the client. Likewise, the Customer Service Provider acts as a client to a backend service provider.

This implementation of IFX uses an Internet protocol suite to provide the communication channel between a client and a server. Internet protocols are the foundation of the public Internet and a private network can also use them. The specific suite of Internet protocols used is HTTP, SSL, TCP and IP. SSL is optional, and it is used to provide security for the communications between the HTTP client and server. SSL 3.0 is recommended, especially for server-to-server application of IFX, since mutual authentication by X.509 certificates can be implemented.

1.2.1 Data Transport

IFX documents can exist outside the context of client/server communication. For example, they can exist as files on persistent storage devices. Nonetheless, this specification focuses primarily on client/server communication of IFX documents.

1.2.1.1 HTTP Post Data Transport

Clients generally use the HyperText Transport Protocol (HTTP) to communicate to an IFX server. The World Wide Web throughout uses the same HTTP protocol. In principle, a Customer Service Provider can use any off-the-shelf web server to implement its support for IFX.

To communicate by means of IFX over the Internet, the client must establish an Internet connection. Clients and servers may also rely on private IP network connections, such as IP encapsulation over a frame relay permanent virtual circuit (PVC), for IFX communication.

Clients use the HTTP POST command to send a request to the previously acquired Uniform Resource Locator (URL) for the desired Customer Service Provider. The URL presumably identifies a Common Gateway Interface (CGI) or other process on a CSP server that can accept IFX requests and produce a response.

The POST identifies the data as being of type text/xml. Use text/xml as the return type as well. Fill in other fields per the HTTP 1.0 spec. Here is a typical request:

```
POST http://www.CSP.com/IFX.cgi HTTP/1.0          HTTP headers
User-Agent:MyApp 5.0
Content-Type: text/xml
Content-Length: 1032

... IFX Document ...
```

A blank line (a carriage return and a linefeed pair—CRLF) defines the separation between the HTTP headers and the start of the IFX XML document. See Chapter 2, “XML Implementation of IFX” for more information about the specific use of XML in the IFX specification.

The structure of a response is similar to the request, with the first line containing the standard HTTP result, as shown next. The content length is given in bytes.

```
HTTP 1.0 200 OK                                   HTTP headers
Content-Type: text/xml
Content-Length: 8732

... IFX Document ...
```

1.2.2 Request and Response Model

The basis for IFX is the request and response model. One or more requests can be batched in a single file. This file typically includes a signon request and one or more service-specific requests. Unless otherwise specified within this specification, a CSP server must process all of the requests and return a single response file. This batch model lends itself to Internet transport as well as other off-line transports. Both requests and responses are plain text files, formatted using a grammar based on Extensible Markup Language (XML). The use of XML allows IFX to evolve over time while continuing to support older clients and servers. In principle, a Customer Service Provider can use any off-the-shelf XML parser/builder to implement its support for IFX.

Here is a simplified example of an IFX request transmission. The indentation in this example is only for readability; white space between elements in an XML document is neither required, nor prohibited. For complete details, see Section 2.2.2 of this document.

```
POST http://www.CSP.com/IFX.cgi HTTP/1.0          HTTP headers
User-Agent:MyApp 5.0
Content-Type: text/xml
Content-Length: 1032

<?xml version="1.0" encoding="UTF-8" ?>           XML header

(start of IFX document)

<?ifx version="1.0.1" oldfileuid="***"            IFX XML PI
newfileuid="***" ?>

<!DOCTYPE IFX PUBLIC "-//IFX//DTD IFX1.0.1//EN"   XML Doctype Declaration
"http://www.ifxforum.org/IFX1.0.1/xml/ifx.dtd"
[private markup]>                                 Private XML Markup

<IFX>                                             IFX request
... IFX requests ...
</IFX>                                            (end of IFX document)
```

The response format follows a similar structure. Although a response such as a statement response contains all of the details of each message, each element is identified using tags.

```
HTTP 1.0 200 OK                                   HTTP headers
Content-Type: text/xml
Content-Length: 8732

<?xml version="1.0" encoding="UTF-8" ?>           XML header

<?ifx version="1.0.1" oldfileuid="***"            IFX XML PI
newfileuid="***" ?>

<!DOCTYPE IFX PUBLIC "-//IFX//DTD IFX1.0.1//EN"   XML Doctype Declaration
"http://www.ifxforum.org/IFX1.0.1/xml/ifx.dtd"    Pointer to public DTD
[private markup]>                                 Private XML Markup

<IFX>                                             IFX response
... IFX responses ...
</IFX>                                            (end of IFX document)
```

1.3 Definitions

1.3.1 User

User refers to the person or entity interfacing with the IFX client to cause it to generate IFX requests.

1.3.2 Client

Client refers specifically to software that generates IFX requests. This may be a personal finance manager, a web browser running locally interactive code (such as with a Java or ActiveX control), a web server, a proxy, or many other possibilities.

1.3.3 Server

Server refers specifically to the software that receives IFX requests, processes them, and generates IFX responses.

1.3.4 Tag

A *tag* the generic name for either a start tag or an end tag. A *start tag* consists of a field name surrounded by angle brackets. An *end tag* is the same as a start tag, with the addition of a forward slash immediately preceding the field name. For example, the start tag for the field named "Foo" looks like this:

```
<Foo>
```

while the end tag for the same field looks like this:

```
</Foo>
```

XML defines empty-element tags, which are used when an element has no content. They have the form `<Foo/>`. Empty-element tags are not allowed in the IFX XML implementation.

1.3.5 Element

An IFX document contains one or more *elements*. An element is some data bounded by a leading start tag and a trailing end tag. For example, an element "Foo," containing data "bar," would look like this:

```
<Foo>bar</Foo>
```

Note that this definition differs slightly from the World Wide Web Consortium (W3C) XML definition of element in that an IFX element must contain data, but may *not* contain other elements. A *W3C XML* element containing other elements is defined in IFX as an *aggregate*. The W3C is the worldwide standards body for web technology.

For more information on the W3C and their standards, see their web site at http://www.w3c.org.

1.3.6 Aggregate

An *aggregate* is a collection of elements and/or other aggregates. An aggregate may not contain any data itself, but rather contains elements containing data, and/or recursively contains aggregates.

1.3.7 Request

A *request* is information sent by the client. An IFX request file is the entire XML file sent by the client, including the XML header. An individual request generally is an aggregate ending in "Rq."

1.3.8 Response

A *response* is information sent by the server. An IFX response file is the entire XML file sent by the server, including the XML header. An individual response generally is an aggregate ending in "Rs."

2 XML Implementation of IFX

This chapter describes various topics involving the implementation of IFX in XML. This includes the basic structure of an IFX document in XML as well as description of how different data types are represented. This chapter also describes how to implement custom tags as well as file based error recovery.

2.1 Structure

As described in Chapter 1, this book explains how IFX is represented in XML and sent over the network via HTTP. The IFX data file consists of standard HTTP header and one IFX XML block. This XML block consists of a signon message and zero or more additional IFX messages wrapped in service wrappers. All IFX data will have the following form:

HTTP headers
IFX XML document

The behavior for both an IFX client and an IFX server has been specified to encourage uniform usage of the specification.

2.1.1 IFX Client

A proper client should separate the components of an IFX request using a single CRLF between each component. A proper request thus has the form:

HTTP headers
CRLF(s)
MIME type information
CRLF(s)
IFX document

2.1.2 IFX Server

An IFX server should expect IFX request components and elements to be separated by the appropriate number of CRLF characters. However, as per W3C recommendations, an IFX server should also accept just a LF as a separator. This behavior is as per the recommendation of the W3C.

http://www.w3.org/Protocols/HTTP/OldClients.html (W3C recommendations)

The text has been included below for ease of reference:

Note: Server tolerance of bad clients
Whilst it is seen appropriate for testing parsers to check full conformance to this specification, it is recommended that operational parsers be tolerant of deviations.

In particular, lines should be regarded as terminated by the Line Feed, and the preceding Carriage Return character ignored.

Any HTTP Header Field Name which is not recognized should be ignored in operational parsers.

It is recommended that servers use URIs free of "variant" characters whose representation differs in some of the national variant character sets, punctuation characters, and spaces. This will make URIs easier to handle by humans when the need (such as debugging, or transmission through non-hypertext systems) arises.

2.1.3 HTTP Headers

The HTTP response returns the standard HTTP result code on the first line. HTTP defines a number of status codes. Servers can return any standard HTTP result. However, IFX servers should expect clients to collapse these codes into the following three cases:

Code	Meaning	Action
200	OK	The request was processed and a valid IFX result is returned.
4xx	Bad request	The request was invalid and was not processed. Clients should report this as an internal error.
5xx	Server error	The server is unavailable. Clients should retry shortly.

The server must return an IFX response file for all 200 and 4*xx* HTTP codes.

An HTTP code of 200 indicates that the file is correct in syntax and data types (including string length). In cases where the file contains one or more incorrect element values, but is correct in syntax and data type, the HTTP code must still be 200, and appropriate IFX response codes must be generated for all messages.

In cases where content is of incorrect data type, the HTTP code must be 4*xx*, and the server must *not* process any messages contained within the file.

In cases where there is a server failure, and the HTTP response code is 5*xx*, the server may still return an IFX response document with a <Status> aggregate, indicating the type of failure.

For more information regarding the specific 4*xx* and 5*xx* codes, consult the IETF HTTP specification.

IFX requires the following HTTP standard headers:

Code	Value	Explanation
Content-type	text/xml	The MIME type for Interactive Financial Exchange XML
Content-length	length	Length of the data after removing HTTP headers

When responding with multi-part MIME, the main type must be multi-part/x-mixed-replace; one of the parts uses text/xml.

2.1.4 IFX XML Document Type Declaration

The contents of an IFX file consist of an XML block, including a processing instruction named "IFX" in the document type declaration.

2.1.4.1 XML Processing Instruction

The XML processing instruction is formatted per the W3C XML standard:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

The version number is the version number of XML.

2.1.4.1.1 Encoding

Encoding defines the text encoding used for character data. The supported values match those of XML 1.0, per W3C recommendations.

2.1.4.2 IFX Processing Instruction

The IFX processing instruction parameters are in the standard *parameter="value"* syntax.

The first parameter must always be *version* with a version number. This entry identifies the contents as an IFX file and provides the version number of the IFX content.

The IFX processing instruction is formatted as follows:

```
<?ifx version="1.0.1" oldfileuid="00000000-0000-0000-0000-000000000000"
newfileuid="00000000-0000-0000-0000-000000000000" ?>
```

The *version* parameter is required. All other parameters are optional.

2.1.4.2.1 Version

Version in the IFX processing instruction specifies the version number of the IFX specification and must be consistent with the Document Type Definition (DTD) used for parsing as specified in the DOCTYPE Declaration. For example, the first IFX version with XML implementation is version 1.0.1.

The *version* tag identifies semantic and/or syntactic changes. In the case of IFX, this corresponds to the specification and DTD. Purely for identification purposes, each change increments the version tag's minor number. If an incompatible change is introduced, such that an older DTD cannot parse the file, the major number should change. See the general discussion of version control, later in this chapter.

2.1.4.2.2 Oldfileuid and Newfileuid

Newfileuid uniquely identifies this request file. The *newfileuid*, which clients supporting file-based error recovery must send with every request file and which servers must echo in the response, serves several purposes:

- Servers may use the *newfileuid* to quickly identify duplicate request files.
- Clients and servers may use *newfileuid* in conjunction with *oldfileuid* for file-based error recovery.
- Servers may use the *newfileuid* to manage the session keys associated with Type 1 application-level security. For more information about security, refer to IFX Message Specification.

Oldfileuid is used together with *newfileuid* only when the client and server support file-based error recovery. *Oldfileuid* identifies the last request and response that was received and processed by the client.

See discussion on file-based error recovery in Section 2.5 of this document.

2.1.4.3 DOCTYPE Declaration

The DOCTYPE declaration is formatted per the W3C XML standard:

```
<!DOCTYPE IFX PUBLIC "-//IFX/DTD IFX 1.0.1//EN"
"http://www.ifxforum.org/IFX1.0.1/xml/ifx.dtd" [private markup goes here]>
```

2.1.4.4 Root Element

The root element of an IFX XML document is the <IFX> tag. It can contain optional namespace attributes for IFX files that have custom elements. e.g.

```
 <IFX xmlns="-//IFX//IFX 1.0.1//EN" xmlns:com.xyz="-//XYZ//XYZ 1.0//EN/-
//IFX//IFX 1.0.1//EN">
```

See discussion on custom element implementation in Section 2.4 of this document.

2.2 XML Details

2.2.1 Compliance

IFX is XML 1.0 compliant. The W3C maintains a specification describing the rules and structure of XML documents. For more information on XML, refer to http://www.w3c.org.

2.2.2 Valid XML Characters

IFX tags that require a value can be set to any sequence of XML characters. To be valid, a value must contain at least one character that is not a blank character. In other words, a value cannot contain only white space. However, all white space within an element is considered to be "important" (i.e., part of the value).

2.2.2.1 Special Characters

Within XML, a few characters must be handled as special characters. To represent a special character, use the corresponding escape sequence.

Character	*Escape sequence*
< (less than)	<
> (greater than)	>
& (ampersand)	&

For example, the string "`AT&T`" encodes "`AT&T`".

2.2.3 Comments Supported

IFX files may contain comments. Comments, in accordance with XML rules, must begin with "`<!--`" and end with "`-->`".

2.3 Data Types

This section describes how the data types used in IFX are represented in XML. Please refer to the IFX Business Message Specification for the semantic and logical description of each time. The following are the supported data types:

Character	Closed Enum
Narrow Character	Open Enum
Binary	Long
Boolean	Identifier
YrMon, Date, Time, DateTime, and Timestamp	Phone Number
Decimal	Universally Unique Identifier (UUID)
Currency Amount	URL

2.3.1 Character

Character fields are identified as a Data Type of "C-*n*," where *n* is the maximum number of allowed Unicode characters. If *n* is absent, the element may contain any amount of characters.

Note: N refers to the number of characters in the resultant string. Depending on the character encoding, each character may be represented by one or more bytes. For example, UTF-8 uses two bytes to encode characters represented in ISO Latin-1 as single bytes in the range 128 through 255 decimal. In addition, characters may be encoded as multi-byte "character entities", and XML encodes the ampersand, less-than symbol, and greater-than symbol as "&", "<", and ">" when they appear as document content. Therefore, an element of type C-40 may be represented by more than 40 bytes in a UTF-8 encoded XML stream.

2.3.2 Narrow Character

Narrow Character fields are identified as a Data Type of "NC-*n*", where *n* is the maximum number of allowed characters. Narrow Character elements must contain only ISO Latin-1 characters.

Note: N refers to the number of characters in the resultant string. Depending on the character encoding, each character may be represented by one or more bytes. For example, UTF-8 uses two bytes to encode characters represented in ISO Latin-1 as single bytes in the range 128 through 255 decimal. In addition, XML encodes the ampersand, less-than symbol, and greater-than symbol as "&", "<", and ">" when they appear as document content. Therefore, an element of type NC-40 may be represented by more than 40 bytes in a UTF-8 encoded XML stream.

2.3.2.1 Identifier

The *Identifier* data type is a specialized version of Narrow Character type. Identifiers have a maximum length of 36. A server generally uses identifiers as database keys.

2.3.2.1.1 UUID

The UUID data type is a specialized version of Identifier, and must contain 36 characters. UUIDs are identifiers that are unique across both space and time, with respect to the space of all UUIDs. The method for computing UUIDs is detailed in the IFX Business Message Specifications. Applications can often obtain conforming UUIDs by calls to the operating system or the run-time environment.

2.3.2.2 URL

A Uniform Resource Locator (URL) is of the Narrow Character data type with a length of up to 1024 characters (NC-1024). URLs are defined in RFC 1738, which is a subset of the Uniform Resource Identifier (URI) specification (RFC 2396). URLs contain only the printable US-ASCII characters 32 through 126 decimal.

2.3.2.3 Phone Number

A Phone Number is of the Narrow Character data type with a length of up to 32 characters (NC-32). The phone number must be formatted as specified in the IFX Business Message Specification.

2.3.3 Boolean

The *Boolean* data type has two states, true or false. True is represented by the literal character 1 (one), while false is represented by the literal character 0 (zero). Unless otherwise specified in this specification, an optional element of type Boolean is implied to be false (0) if it is absent.

2.3.4 Numeric

Numeric data types accept the ISO-646 digits (0–9), period (.), plus (+), and minus (-) characters. The period is only permitted as a separator between the integer and the fractional amount (i.e., a decimal point), for type *Decimal* only. Thousands separators are not allowed.

All numeric formats use a leading sign. Negative numbers use a minus sign (-), while positive numbers use a plus sign (+). Absence of a sign implies a positive number.

2.3.4.1 Long

The *Long* data type is an Integer expressed as a Base-10, ASCII character set string representation of a 32-bit signed integer in the range -2147483648 to +2147483647. Elements of type Long do not permit a decimal point.

2.3.4.2 Decimal

The *Decimal* data type indicates a numeric value that is up to fifteen (15) digits in length, excluding any punctuation (e.g. sign or decimal), is not restricted to integer values and has a decimal point that can be placed anywhere from the left of the leftmost digit to the right of the rightmost digit. Absence of a decimal point implies one to the right of the rightmost digit (i.e., an integer).

Example: +1234567890.12345 is acceptable, while 12345678901234567 is not.

The *Decimal* data type is always expressed as a Base-10, ASCII-character-set string.

2.3.5 Binary

The binary data type is implemented as an aggregate containing up to three elements:

Tag	*Type*	*Usage*	*Description*

Tag	Type	Usage	Description
<ContentType>	Open Enum	Optional	Specified in IETF RFC 2046. Defined values: hex, base64. Default if not present: hex
<BinLength>	Long	Required	Identifies the size of the binary data in number of bytes before encoding.
<BinData>	NC	Required	Encoded binary data.

2.3.6 Dates, Times, and Time Zones

There is one format for representing dates, times, and time zones. The complete form is:

*YYYY-MM-DD*T*HH*:*mm*:*ss*.*ffffff*±*HH*:*mm*

where all punctuation and the "T" are literal characters; "*YYYY*" represents a four-digit year; "*MM*" represents a two-digit month; "*DD*" represents a two-digit date; the first "*HH*" represents a two-digit, 24-hour format hour; the first "*mm*" represents a two-digit minute; "*ss*" represents a two-digit second; and "*ffffff*" represents fractional seconds, and may be of any length.. The second "*HH*" and "*mm*" describe the time zone offset from coordinated universal time (UTC), in hours and minutes, respectively. The "±" can be either a "+" or a "-" depending on whether the time zone offset is positive or negative.

> ***Note:*** *IFX requires the recipient of an IFX message to store the value to at least the same precision as sent, or milliseconds, which ever is less precise.*

2.3.6.1 DateTime

Tags specified as type *DateTime* and generally starting with the letters "DT" accept a fully formatted date/time/timezone string. For example, "1996-10-05T13:22:00.124-5:00" represents October 5, 1996, at 1:22 and 124 milliseconds PM, in Eastern Standard Time. This is the same as 6:22 PM Coordinated Universal Time (UTC).

Several portions of a DateTime element are optional. The following table describes the optional components and the meaning if they are absent:

Component	Meaning if absent
±*HH*:*mm* (time zone offset)	+00:00 (UTC)
T*HH*:*mm*:*ss*.*ffffff*±*HH*:*mm* (time component)	T00:00:00+00:00 (midnight, UTC)
:ss.ffffff (seconds and fractional seconds)	:00.000000 (zero seconds)
.ffffff (fractional seconds)	.000000 (zero fractional seconds)

> ***Note:*** *times zones are specified by an offset, which defines the time zone. Valid offset values are in the range from -12:59 to +12:59, and the sign* **is required**.

Take care when specifying an ending date without a time. If the last transaction returned for a bank statement download was January 5, 1996 10:46AM, Eastern Standard Time, and if the <EndDt> was given as just January 5, the transactions on January 4 after 7:00PM, Eastern Standard Time, (noon minus the five-hour offset) would be resent. If results are available only daily, then just using dates and not times will work correctly.

2.3.6.2 Date

Tags specified as type *Date* accept dates in the *YYYY-MM-DD* format.

2.3.6.3 YrMon

Tags specified as type *YrMon* accept years and months in the *YYYY-MM* format.

2.3.6.4 Time

Tags specified as type *Time* and generally ending with the letters "TM" accept times in the following format:

hh:*mm*:*ss*.*fffffff*±*HH*:*mm*

The seconds, milliseconds and time zone offset are still optional, and default as specified in Section 2.3.6.1.

2.3.7 Currency Amount

The Currency Amount data type is implemented as an aggregate containing up to four elements, as specified in the Business Message Specification.

2.3.8 Definition of Data Types in the DTD

In addition to defining the structure of the IFX XML document, the IFX DTD also provides type information for each element. The type information is provided in the form of an attribute list for the element. The attributes defining the types have a "dt" Namespace prefix. The first attribute is the name of the type and the second attribute is the attribute of the type (e.g. maximum length of a string type). Here are some examples:

```
<!-- Datatype Namespace declaration -->
<!ATTLIST IFX xmlns:dt CDATA #FIXED "urn:schemas-microsoft-com:datatypes">

<!-- Element and Type definition for Addr1 -->
<!ELEMENT Addr1 #PCDATA>
<!ATTLIST Addr1                                    dt:type NMTOKEN #FIXED
"string"

                                                          dt:maxlength CDATA
#FIXED "32">
<!--#ENTITY % Addr1 #DataType(A-32)-->

<!-- Element and Type definition for BillStatusCode -->
<!ELEMENT BillStatusCode #PCDATA>
<!ATTLIST BillStatusCode dt:type NMTOKEN #FIXED "string"
 dt:enumeration CDATA #FIXED "New Viewed Delivered Withdrawn Retired
Undelivered">
<!--#ENTITY % BillStatusCode #Enum("New", "Viewed", "Withdrawn", "Retired",
"Undelivered")-->
```

> ***Note:*** *An OFX 1.5.1 style of type definition (in comments) is also included to allow OFX software to easily migrate to IFX.*

The following table shows how the different types are declared in the DTD using attributes and the comment style:

IFX Type	*Type Declaration using Attributes*	*OFX-style Type Declaration*
Character (C-n)	dt:type="string" dt:maxLength="n"	DataType(A-n)
Narrow Character	dt:type="string" dt:maxLength="n"	DataType(A-n)
Identifier	dt:type="string" dt:maxLength="32"	DataType(A-32)
UUID	dt:type ="uuid"	DataType(A-36)
Boolean	dt:type ="boolean"	DataType(BOOL)
Long	dt:type ="integer"	DataType(I-n)
Decimal	dt:type ="decimal"	DataType(N-n)

DateTime	dt:type =“dateTime”	DataType(DATE)
Date	dt:type =“date”	DataType(DATE)
YrMon	dt:type =“yrMon”	DataType(DATE)
Time	dt:type =“time”	DataType(TIME)
Timestamp	dt:type =“dateTime”	DataType(DATE)
URL	dt:type =“uri”	DataType(A-1024)
Closed Enum	dt:type =“string” dt:enumeration =“value1, value2, …”	Enum(“value1”, “value2”, …)
Open Enum	dt:type =“string”	dt:type =“string”
Binary	Aggregate. See Section 2.3.5.	N/A
Currency Amount	Aggregate. See Section 2.3.7.	DataType(N-n)
Phone Number	dt:type =“string” dt:maxLength =“32”	DataType(A-32)

2.4 XML Implementation of IFX Extensions

An organization that provides a customized client and server that communicate by means of IFX might wish to add new requests and responses or even specific elements to existing requests and responses. To ensure that each organization can extend the specification without the risk of conflict, IFX defines a style of tag naming that lets each organization have its own name space. Please refer to the IFX Business Message Specification, Section 2.7.1, for detail discussions of IFX Extensions.

IFX Extensions are implemented in XML by the use of Namespace. For more information about Namespace in XML, refer to http://www.w3.org/TR/1999/REC-xml-names-19990114.

2.5 File-based Error Recovery

File-based error recovery allows an IFX client to retrieve a response file that was previously generated by the IFX server. This provides an alternative way for error recovery from synchronization or audit. In file-based error recovery, the servers keep a copy of the entire response file they last sent. Clients requesting that servers prepare for error recovery generate a universally unique ID for each file they send. In the IFX headers, there are two tags associated with error recovery:

- **oldfileuid –** UID of the last request and response that was successfully received and processed by the client
- **newfileuid –** UID of the current file

Servers use the following rules:

- If *newfileuid* is absent, the client is not requesting file-based error recovery for this session. The server does not need to save the response file. If *newfileuid* is absent and *oldfileuid* matches a previous request file (see below), the client may be ending use of file-based error recovery.
- If *newfileuid* matches a previous request file, the client is requesting error recovery. The server should send the matching saved response file.

  Note: *If* newfileuid *matches a previous request file's UID then the request file identified by the* newfiluid *must contain exactly the same set of messages as the previous request file. Servers can reject the file if it contains new or modified messages. In particular, clients should disallow <CustPswdModRq> messages during error recovery.*
- If *newfileuid* is present and does not match a previous request file, the client is preparing for error recovery. The server should save the response file in case the data does not reach the client.

- If *oldfileuid* is absent, the server should not search for a response file to delete. Clients should initiate file-based error recovery by omitting *oldfileuid* and providing a *newfileuid* set to a unique value.
- If *oldfileuid* matches a file saved on the server, then *oldfileuid* is a file that the client has successfully processed and the server can delete it.
- If *oldfileuid* is present and does not match a previous request file, the server should ignore the presence of this processing instruction. Either the server has purged the associated request file without explicit request from the client or the client is requesting error recovery with identical headers to the initial request attempt (in which case *newfileuid* should match a previous request file).

  Note: *While it may indicate a client error for* oldfileuid *and* newfileuid *to hold identical values, the server must ignore the* oldfileuid*. Earlier rules in this list detail how the server should handle the request file (based solely upon the* newfileuid *value).*

A server need not save more than one file per client data file thread (history of *newfileuid* values), but because of possible multiclient or multidata file usage, it might need to save several files for a given user. A server should save files for as long as possible, but not indefinitely (2 months is recommended). If an error recovery attempt comes after the corresponding error recovery file is purged, the server will not recognize the request as an attempt at error recovery. The server would simply process it as a new request. In this case, the server should recognize duplicate <RqUID>s for client-initiated work, such as payments, and then reject them individually. Server-generated responses would be lost to the client. A server should not save a response file when it is useless to do so. Specifically, the Accept server should not save a response file when the request fails parsing or when the request was rejected due to a <SignonRq> problem (for example, invalid <CustId>).