



Research Centre Europe

Xerox Linguistic Suite

Programmer's Guide

© 2005 by The Document Company Xerox and Xerox Research Centre Europe. All rights reserved.

Copyright protection claimed includes all forms and matters of copyrightable material and information now allowed by statutory or judicial law or hereinafter granted, including without limitation, material generated from the software programs which are displayed on the screen such as icons, screen displays, looks, etc.

Printed in France

Xerox®, The Document Company®, and all Xerox products mentioned in this publication are trademarks of Xerox Corporation.

Table of Contents

1	About This Manual	4
2	Introduction	6
3	Setting Up the Development Environment	7
3.1	Structure of the XF C SDK	7
3.2	Paths to Include Files and Libraries	7
3.3	Setting Environment Variables	8
3.4	Compiling XF Applications	8
3.5	Using the Example Applications	9
4	Overview of General Domain Components	11
4.1	Naming conventions	11
4.2	Memory allocation	12
4.3	Error and messages	12
4.4	String c-classes	13
4.5	Collection c-classes	15
5	Text Processing Modules	18
5.1	XFModule base c-class	18
5.2	XFModuleManager c-class	18
5.3	Result and metadata	19
5.4	Anatomy of Text Processing Module	19
6	Modules available	21
6.1	Input/Output modules	21
6.2	Linguistic modules	22
6.3	Utility modules	25
7	Using XF Low level API	27
7.1	XF FST low level API	27
7.2	HMM low level API	30

1 About This Manual

The Xerox Linguistic Suite is a toolkit for developing custom linguistic applications. The Xerox Linguistic runtime engine can be used to transform, normalize, and extract information from text.

This overview introduces the Xerox linguistic services and the modes in which Xerox Linguistic applications can operate.

This manual describes in detail the C Software Development Kit (SDK) provided with the Xerox Linguistic Suite for creating custom language processing application

Audience

This manual is designed for developers in the fields of computing and linguistics who understand linguistic terminology and C programming concepts.

Convention used

This overview uses the following style conventions:

- ◆ **Monospaced font:** this typeface is used for any text that appears on the computer screen or text that you should type. It is also used for file names, functions, and examples.
- ◆ *Monospaced italic font:* this typeface is used for any text that serves as a placeholder for a variable. For example, `dataType` indicates that the word `dataType` is a placeholder for an actual data type, such as `CString`.
- ◆ [Internal cross references](#): this format is used to indicate cross-references within the manual. If you are working on an electronic copy of the manual, click the cross-reference to go directly to the section it references.
- ◆ Install paths are based on the default install paths. For example, on Windows the path to the include directory is `<install_dir>\c-fsm\include`. The path to the include directory on Solaris is `<install_dir>/c-fsm/include`.

What This Manual Contains

This overview contains the following sections:

- ◆ [Introduction](#)
Provides an overview of the development process, gives system requirements, and lists the languages supported by the Xerox Linguistic Suite.
- ◆ [Setting Up the Development Environment](#)
Describes how to prepare a development environment for creating applications using the Xerox Linguistic SDK.
- ◆ [Overview of General Domain Components](#)
Describes the general domain components used by XF.
- ◆ [Text Processing Modules](#)
Describes the functions of text processing module and how to create its own one.
- ◆ [Modules available](#)
Describes the modules available in the XF toolkit.
- ◆ [Using XF Low level API](#)
Describes how to use the Low level API available in the Xerox Linguistic

SDK, such as the loading a transducer, looking up a word in a cascade of transducers or calling the part of speech disambiguation service.

Required Reading

Before you create your own custom application using the XF C API, read the Xerox Linguistic Suite Overview, a manual that describes the linguistic services available.

Other Documentation

In addition to this manual, the documentation set for the Xerox Linguistic Suite includes the following manuals:

- ◆ Xerox Linguistic Suite Overview: provides a basic overview of the Xerox linguistic services.
- ◆ Xerox Linguistic Suite Installation Guide: provides procedures for installing the Xerox Linguistic Suite.
- ◆ XF API Reference Manual: provides a complete reference to all of XF public functions.
- ◆ Xerox Linguistic Reference Guide: provides information about the languages supported by the Xerox Linguistic Suite, including the tags associated with each language.

2 Introduction

This manual contains all the information you need to create an application that uses the XF SDK.

Application Development Overview

Before you can develop a XF application, you must first set up your development environment so that you can successfully compile and debug your application.

Once you have set up your development environment, you can begin work on your XF application. Every application that use XF must perform the following basic operations:

- ◆ Initialize the XF toolkit
- ◆ Instantiate and connect the required linguistic services modules
- ◆ Run the application
- ◆ Clean up the allocated components

This manual describes first how to set up your development environment and then how to use the various features of XF to create your custom natural language processing system.

System Requirements

To develop a Windows application that uses the XF SDK, you must work on Windows 2000 or XP and compile using the Microsoft Visual C++ Compiler version 6.0 or .Net 2003.

For Solaris applications, use Sun Solaris 8 or 9 and compile using the GNU GCC C Compiler or Sun WorkShop/Forte C Compiler.

For Linux applications, compile using the GNU GCC C Compiler.

3 Setting Up the Development Environment

This chapter describes how to prepare a development environment for creating custom XF applications. It contains the following sections:

- Structure of the XF C SDK
- Paths to Include Files and Libraries
- Setting Environment Variables
- Compiling XF Applications
- Using the Example Applications

3.1 Structure of the XF C SDK

The XF C SDK is composed of the following libraries:

- ICU: this library provides a portable implementation for converting text from/to Unicode.
The source code of the library can be obtained from <http://ibm.com/software/globalization/icu>
- Expat¹: a XML parser library written in C. The source code of the library can be obtained from <http://expat.sourceforge.net/>.
- FST: this library is Xerox proprietary.

3.2 Paths to Include Files and Libraries

The XF installation² tree appears as follows by default:

Directory	Description
c-fsm-<VERSION>	Top directory of the c-fsm installation
c-fsm	Top directory of the c-fsm sources
include	Directory containing the include files
ix86-linux2.6-gcc3.4	Top directory for the linux specific files
bin	Directory for the linux specific programs
lib	Directory for the linux specific libraries
ix86-windowsxp-msvc71	Top directory for the windows specific files
bin	Directory for the windows specific programs
lib	Directory for the windows specific libraries
doc	Top directory for the documentation
xf	Documentation of th XF API
src	Top directory for the sources
rules	Makefile rules
fst	fstbase runtime
tokenize	fstapply runtime
disamb	Standalone part of speech disambiguation tool
fst_streamer	All-in-one Linguistic runtime
xf	New version of the fst_streamer
c-fsm-import	Top Directory of the dependencies

The paths to the XF include files and libraries vary depending upon the operating

¹ Expat is only necessary to use the xfx sample application.

² Depending on the agreement, the package may not contain the sources of the runtime.

system you are using. The following sections describe where to find the include files and libraries on Windows and Solaris environments.

3.2.1 Paths on Windows

All of the include files are installed in a sub directory of the
`<install_dir>\c-fsm\include`.

By default, all of the libraries (all `.lib` and `.dll` files) are installed in the
`<install_dir>\c-fsm\ix86-windowsxp-msvc71\lib` directory.

3.2.2 Paths on Solaris

By default, all of the include files are installed in a sub directory of the
`<install_dir>/c-fsm/include` directory.

By default, all of the libraries (all `.so` files) are installed in the
`<install_dir>/c-fsm/sparc-solaris9-forte6u2/lib` directory.

3.2.3 Paths on Linux

By default, all of the include files are installed in a sub directory of the
`<install_dir>/c-fsm/include` directory.

By default, all of the libraries (all `.so` files) are installed in the
`<install_dir>/c-fsm/ix86-linux2.6-gcc3.4/lib` directory.

3.3 Setting Environment Variables

The XF SDK depends on the following environment variables

- ◆ **CL_VER:** Windows only. Defines the compiler name. This is used for generated a unique directory name where compiler-dependent object, libraries and programs are generated. Recommended values are:
 - ◆ **msvc6** for Microsoft® Visual C++ version 6 (directory name is then `ix86-windowsxp-msvc6`)
 - ◆ **msvc71** for Microsoft® Visual C++ version .Net 2003 (directory name is then `ix86-windowsxp-msvc71`)
- ◆ **PATH:** Windows only. Defines the path to the programs and the Windows DLLs.
- ◆ **LD_LIBRARY_PATH:** Linux&Solaris only. Defines the path to the shared libraries. See the man page for more information.

3.4 Compiling XF Applications

How you compile and debug applications built with the XF C SDK depends upon your development environment. This section contains information on the following:

- ◆ Compiling on Windows
- ◆ Compiling on Linux & Solaris

3.4.1 Compiling On Windows

Whenever you compile, you should set the same flags as the one set when compiling the `fst` and `xf` libraries. In particular, set the multithread global flags as follows:



/MD

The following table lists all of the non-xerox libraries, flags, and dependencies for compiling on Windows:

Library	Include Sub Directory	DLL names	Flags	Dependencies
icu	import/icu/3.2/include	icuuc32.dll		
expat	import/expat/1.95.8/include	libexpat.dll		

The following table lists all of the xerox libraries, flags, and dependencies for compiling on Windows:

Library	Include Sub Directory	DLL names	Flags	Dependencies
fstbase	fst	fstbase.dll	/D"_FSTBASEDLL"	
fstcalc	fst	fstcalc.dll	/D"_FSTCALCDLL"	
fstapply	tokenize	fstapply.dll	/D"_FSTAPPLYDLL"	fstbase
xf	xf	xf.dll	/D"_XFDLL"	fstbase, fstapply

3.4.2 Compiling On Linux & Solaris

The following table lists all of the non-xerox libraries, flags, and dependencies for compiling on Solaris and linux:

Library	Include Sub Directory	Library names	Dependencies
icu	import/icu/3.2/include	libicuuc.so	
expat	import/expat/1.95.8/include	libexpat.so	

The following table lists all of the xerox libraries, flags, and dependencies for compiling on Solaris and linux:

Library	Include Sub Directory	Library names	Dependencies
fstbase	fst	libfstbase.so	
fstcalc	fst	libfstcalc.so	
fstapply	tokenize	libfstapply.so	fstbase
xf	xf	libxf.so	fstbase, fstapply

3.5 Using the Example Applications

You can use the example applications shipped with XF to create your own custom applications. The sample applications are located in the following folders:

➤ On Windows: <install_dir>\c-fsm\src\xf\samples

➤ On Solaris: <install_dir>/c-fsm/src/xf/samples

The folder contains the following subdirectories:

- postag: an example of use of the XF low level API.
- xftag: an example of a simple XF application that does not make use of the module manager and create module by calling their initialization function directly
- xftest: an example of a XF application that make use of the module manager.
- xfx: an example of a XF application that creates a XF runtime by reading its description from a XML file.

Also, see the `README.txt` file provided in each subdirectory for information about the sample application.

**TODO: Write
Visual C projects**

Each example contains a Makefile and a Microsoft Visual C project for compiling them.

4 Overview of General Domain Components

This section describes the general domain components available in the XF library.

This section contains information about the following classes:

- ➡ Naming conventions
- ➡ Memory allocation
- ➡ String c-classes
- ➡ Collection c-classes: Vector, Hashtable and Iterators

4.1 Naming conventions

Although, the library is written in pure C, some concepts from C++ have been borrowed and simulated in C.

4.1.1 C-Classes

To simulate classes in C, all structures have all the same declaration structure and all methods follows the same naming and calling conventions.

From now on, c-style classes, objects and methods are called c-class, c-objects and c-methods.

- ➡ Definition: All structures have 3 type names:

```
1 typedef struct _<NS><CLASSNAME>Rec { ... };
2 typedef struct _<NS><CLASSNAME>Rec <NS><CLASSNAME>Rec;
3 typedef typedef struct _<NS><CLASSNAME>Rec *<NS><CLASSNAME>;
```

Where <NS> is the namespace (i.e. XF) and <CLASSNAME> is the class name.

Since pointers are used most of the time, names like XF<CLASSNAME> are the only type used.

- ➡ c-methods: like in C++, all functions takes a pointer to the object they apply on. Moreover, a c-method name starts with the c-class name and is followed by the method name. For example:

```
size_t XFStringGetLength(XFString string);
```

is the method "GetLength()" of the c-class String of the package XF.

4.1.2 new, delete

All classes have a XF<Class>New() which is similar to the new <Class>() of C++. The counterpart of the C++ delete object is the function XF<Class>Delete(). All c-classes also expose 2 more functions: XF<Class>Init() which is similar to the C++ constructor and XF<Class>Release() which is similar to the C++ destructor.

4.1.3 Class initialization

All c-class have a XF<Class>ClassInit() and XF<Class>ClassRelease() functions that are called by the XFInit() and XFRelease() functions to initialize and release internal structures.

4.1.4 Inheritance and virtual functions

Inheritance would be difficult (if possible) to simulate in C. nevertheless, a small set of virtual functions can be simulated by hand. This is done by the XFModule c-class.

4.2 Memory allocation

Memory allocation should be done using the XF memory allocation functions. They are calling the similar functions of the C library with additional tests:

- ◆ After having allocated memory, the pointer returned is checked and if it is NULL, an error message is printed.
- ◆ When freeing memory, the pointer to free can be NULL.

The main functions are:

- ◆ `XFMalloc()`
- ◆ `XFRealloc()`
- ◆ `XFFree()`

4.3 Error and messages

A simple yet efficient error and message system is used to report errors and format messages to be presented the user.

The error system stores all messages (information, warning, error, ...) until `XFStatusClear()` is called.

4.3.1 Status code

Every functions returns either a result or a status code (`XFStatus`). When a result is returned, it must be checked if it is NULL. In such case the error status can be retrieved with the `XFErrorGetLastError()` function.

A predefined list of error codes are defined in `error.h`. It covers general errors like memory allocation problems, file errors, etc

The last status and message can be reset with the `XFStatusClear()` function.

4.3.2 Messages

A predefined message is associated with each status code. There are written in English. Each status code ends with a series of letters to reminds the programmers the list of arguments the associated message requires:

- ◆ S: Argument is a `char *` string
- ◆ I: Argument is a integer
- ◆ F: Argument is a float

To format a message, call the `XFFormatMessage()` or `XFMsg()` function with the status code and the needed parameter. `XFMsg()` returns the status code itself. For example, when a file can not be open, the message is formatted using:

```
1 status = XFMsg(XFFileCannotOpenS, "/path/to/file");
```

The formatted message is available with the `XFErrorGetLastErrorMessage()` function.

```
2 fprintf(stderr, "%s\n", XFErrorGetLastErrorMessage());
```

A level can be added to the message. Available levels are:

- ◆ `XFDebug(status)`: Add the prefix [DEBUG] to the last message
- ◆ `XFInfo(status)`: Add the prefix [INFO] to the last message

- ➡ `XFWarning(status)`: Add the prefix `[WARNING]` to the last message
- ➡ `XFError(status)`: Add the prefix `[ERROR]` to the last message
- ➡ `XFInternalError(status)`: Add the prefix `[INTERNAL ERROR]` to the last message followed by the name of the source file (`__FILE__`) and line (`__LINE__`) where this error occurred.

A typical use :

```
1 status = XFError(XFMsg(XFFileCannotOpenS, "/path/to/file"));
```

or

```
1 return XFWarning(XFMsg(XFFileCannotOpenS, "/path/to/file"));
```

4.4 String c-classes

This section describes the string and text related c-classes commonly used in XF applications. It contains information about the following classes:

- ➡ `XFString` c-class
- ➡ `XFUnicodeString` c-class

4.4.1 XFString c-class

XF uses the `XFString` c-class to manipulate c-string (`char*`). This section describes the major member functions of this c-class in more detail.

For more information about the functions of the `XFString` c-class, refer to the XF Reference Manual.

Constructors

There are many different constructors for creating a `XFString` c-object. Some of the most frequently used constructors follow:

- ➡ `XFStringNew()`: Creates an empty string.
- ➡ `XFStringNewFromCString(const char *str)`: Constructs a `XFString` from a `char *` string (encoded in ISO 8859-1).
- ➡ `XFStringCopy(const XFString str)`: copy a `XFString`.

General Methods

The `XFString` c-class has the following general methods:

- ➡ `XFStringGetLength()`: returns the length of the string.
- ➡ `XFStringGetSubString()`: returns a substring of this string.

Conversion Methods

The `XFString` c-class has the following conversion methods:

- ➡ `XFStringToCString()`: returns the pointer to the internal `char` representation for interfacing with functions taking a `const char*` as parameter.

Search and Comparison Methods

The `XFString` c-class has the following search and comparison methods:

- ◆ `XFStringCompareCString()`, `XFStringNCompareCString()`, `XFStringCompareString()`: `XFStringNCompareString()`: compares the `XFString` with another string and returns an integer less than, equal to, or greater than zero. The “CString” version compares with a `char*` string whereas the “String” compares with a `XFString`. The “N” version compares only the first N characters.
- ◆ `XFStringFindForwardCString()`, `XFStringFindForwardString()`: returns the first occurrence of a string starting at the specified input position.
By default, the comparison is case sensitive, meaning it distinguishes between uppercase and lowercase characters.

Modification Methods

The `XFString` c-class has the following modification methods:

- ◆ `XFStringSetCString()`, `XFStringSetString()`: clear the string and set the new value with the given `char*` or `XFString` string.
- ◆ `XFStringReplaceCString()`, `XFStringReplaceString()`: replaces all or part of a string by all or part of another string.
- ◆ `XFStringAppendCString()`, `XFStringAppendString()`: appends a character or a string to this string.

4.4.2 XFUnicodeString c-class

XF uses the `XFUnicodeString` c-class to manipulate unicode string (`UChar*`). This c-class encapsulate the calls the an external library that perform the operation on Unicode string as well as converting from/to other character set. This section describes the major member functions of this c-class in more detail.

For more information about the functions of the `XFUnicodeString` c-class, refer to the XF Reference Manual.

Constructors

There are many different constructors for creating a `XFUnicodeString` c-object. Some of the most frequently used constructors follow:

- ◆ `XFUnicodeStringNew()`: Creates an empty string.
- ◆ `XFUnicodeStringNewFromCString(const char *str)`.
Constructs a `XFUnicodeString` from a `char *` string encoded in codeset given in parameter.
- ◆ `XFUnicodeStringNewFromString(const XFUnicodeString str)`: copy a `XFUnicodeString`.

General Methods

The `XFUnicodeString` c-class has the following general methods:

- ◆ `XFUnicodeStringLength()`: returns the length of the string.
- ◆ `XFUnicodeStringNewSubString()`: returns a substring of this string.

Conversion Methods

The `XFUnicodeString` c-class has the following conversion methods:

- ◆ `XFUnicodeStringToXFString()`: convert this Unicode string to a c-style string encoded in the codeset given in parameter.

- ➡ `XFUnicodeStringGetInternalString()`: returns the pointer to the internal char representation for interfacing with functions taking a `UChar*` as parameter.
- ➡ `XFUnicodeStringSetDefaultInputCodeset()`, `XFUnicodeStringSetDefaultOutputCodeset()`: set the default encoding to use when function requiring an codeset as parameter are given a NULL codeset. The “Input” version set the default codeset to use when converting to Unicode whereas the “Output” version set the default codeset to use when converting from Unicode.

Search and Comparison Methods

The `XFUnicodeString` c-class has the following search and comparison methods:

- ➡ `XFUnicodeStringCompare()`: compares the `XFUnicodeString` with another string and returns an integer less than, equal to, or greater than zero.
- ➡ `XFUnicodeStringFindForward()`: returns the first occurrence of a string starting at the specified input position. By default, the comparison is case sensitive, meaning it distinguishes between uppercase and lowercase characters.

Modification Methods

The `XFUnicodeString` c-class has the following modification methods:

- ➡ `XFUnicodeStringReplaceCString()`, `XFUnicodeStringReplaceString()`: replaces all or part of a string by all or part of another string.
- ➡ `XFUnicodeStringAppendChar()`, `XFUnicodeStringAppendCString()`, `XFUnicodeStringAppendString()`, `XFUnicodeStringAppendSubString()`: appends a character, a string or a sub-string to this string. The `CString` version first convert the string parameter to the encoding given in parameter.

4.5 Collection c-classes

The XF library makes heavy use of collection classes. Therefore it provides an implementation of a Vector and Hashtable to have a standalone, powerful and easy to use functionality.

4.5.1 Vector c-class

The XF library provides its own implementation of a vector c-class that can store any type of object. To achieve this, it needs to call function that depend on the real type of objects it contains.

IMPORTANT: The `Insert()`, `Remove()`, `GetElement()`, `copy()` and `delete()` functions take as parameter or return pointers to the type of object stored. For example, if the Vector stored `XFString` object, the `GetElement()` function will return a `XFString*` and `Insert()` must be called also with a `XFString*` as parameter. It is heavily recommended to create a specialized version of the Vector for object frequently stored in Vector. See `XFStringVector` and `XFUnicodeStringVector` include file for an example of specialization of the Vector c-class.

Copy and delete functions

Since a `Vector` can store any type of object, it needs to call function that depend on the type of objects stored. These functions must be set after having created a `Vector` c-object by calling the `XFVectorSetFunctions()` function.

The `Vector` c-class needs:

- ◆ a “`element_size`” parameter to calculate the size of the memory to allocate to store an element.
- ◆ a copy function called when inserting an element (`XFVectorInsert()`). The default copy function is to copy “`element_size`” byte from the pointer passed as parameter.
- ◆ a delete function called when removing an element (`XFVectorRemove()`) or deleting all the elements (`XFVectorDelete()`). The default delete function does nothing.

4.5.2 Hashtable c-class

The XF implementation of the hashtable functionality can also store any type of object. To achieve this, it needs to call function that depend on the type of objects stored.

The `Hashtable` behavior is more classic than the `vector` c-class because it stores pointer to key and value. Therefore, there is no need of a key size or value size. Moreover the `Insert()` and `Get()` method take pointer to key/value as arguments.

Copy, delete, hash and compare functions

Since `Hashtable` can store any type of object., it needs to call function that depend on the type of objects stored. These functions are set by calling the `XFHashtableSetFunctions()` function.

The `hashtable` c-class needs:

- ◆ a key-copy and value-copy functions called when inserting an key-value pair (`XFHashtableInsert()`). The default copy function is to copy the key and value pointers.
- ◆ a delete-key and delete-value functions called when removing an element (`XFHashtableRemove()`) or deleting all the elements (`XFHashtableDelete()`). The default delete function does nothing.
- ◆ a hash function that compute a hash number from a key. The default hash function assumes that the type of the key is a `char * string`.
- ◆ a compare function that compares 2 keys having the same hash number. The default compare function assumes that the type of the key is a `char * string` and calls the `strcmp()` function.

4.5.3 Iterators

The iteration on elements of a `Vector` or `Hashtable` is always done by using an iterator. Such iterators can be created by calling the `GetIterator()` function with the c-object on which to iterate as parameter. All iterators have the following methods:

- ◆ `Delete()`: Delete the iterator
- ◆ `HasNext()`: returns `true` if the collection has more elements or `false` if the iterator has reached the end of the collection.

- ➡ `Next()`: Move iterator to next element in the collection. `XFVectorNext()` returns also the next element. For a hashtable use `XFHashtableGetKey()` and `XFHashtableGetValue()`.
- ➡ `GetElement()`, `GetKey()` or `GetValue()`: get the current element, key or value the iterator is pointing at.

In addition, the `XFVectorIterator` has a `XFVectorIteratorRemove()` function that allows to remove any element from a vector while iterating on it. The `XFHashtableIterator` does not offer such a function.

5 Text Processing Modules

The main goal of the XF toolkit is to create a Text Processing engine composed of several Text Processing modules connected together. The engine is then run by simply calling the `Process()` function of the last module.

Therefore all text processing components are handled as `XFModule` by the top-level module functions, giving a simple, flexible and powerful system.

This section contains information about the following classes:

- ◆ `XFModule` base c-class
- ◆ `XFModuleManager` c-class
- ◆ Anatomy of Text Processing Module

5.1 XFModule base c-class

A `XFModule` c-class stores the following information:

- ◆ A pointer to the module class definition. It contains the functions specific to a particular module. This is similar to the C++ virtual functions.
- ◆ A pointer to the previous module it will get new data from. The only exception is the first module that takes its data from a file, a string or any data repository.
- ◆ The metadata and result generated by the module
- ◆ The metadata and result generated by the previous module
- ◆ The state of the module

The module class definition (`XFModuleClass`) contains the following members:

- ◆ `release`: the function to call to release the module allocated by the module. This is similar to the C++ virtual destructor
- ◆ `getName`: the function to call to get the name of the module.
- ◆ `setParameters`: the function to call to set some parameters of the module.
- ◆ `process`: the main function that get data, process them and generates new data and metadata.

5.2 XFModuleManager c-class

The `XFModuleManager` is a singleton that helps the creation of a module from a string instead of having to call directly the `New()` function of such a module.

Therefore, each module must call the function

`XFModuleManagerSingletonAddModule()` with the name of the module and the creation function as parameter. This must be done in the `ModuleClassInit()` function.

To create a module, calls the `XFModuleManagerSingletonCreateModule()` function with the name of the module to create and its parameters.

5.3 Result and metadata

Each module process data and pass the result of the processing to the calling module (i.e. The next one in the processing chain). Each result is associated with metadata that specify what is the function, or type of this result. The format of the metadata string is an XML-like format. Each module should append or replace metadata to the metadata string parameter. Therefore the result and metadata can be seen as 2 synchronized streams.

For example the disambiguation module receives the surface form of a word to disambiguate with the attached metadata after the first call to the module performing morphological analysis.

```
1 result: "The"
2 metadata: "<file filename='sample.txt'><sentence id='1'><lexeme-
list><lexeme id='1'><surface_form>"
```

Then the different analysis for this surface form are returned with the following metadata:

After the second call to the module performing morphological analysis:

```
1 result: "The+PROP"
2 metadata: "<file filename='sample.txt'><sentence id='1'><lexeme-
list><lexeme id='1'><sense id='1'>"
```

After the third call to the module performing morphological analysis:

```
1 result: "the+DET"
2 metadata: "<file filename='sample.txt'><sentence id='1'><lexeme-
list><lexeme id='1'><sense id='2'>"
```

etc..

A special module `format-xml` takes the metadata and generate a XML result. After this module, there is no more value in the metadata stream.

5.4 Anatomy of Text Processing Module

Text processing module definition and initialization

A Text Processing module is defined by “deriving” a new c-class from the `XFModule` c-class. This is done have a `XFModuleRec` as first member of the c-class followed by the specific data needed by the text processing module.

```
1 typedef struct _XF<CLASSNAME>Rec {
2     XFModuleRec super;
3     /** module specific data */
4     ...
5 };
```

Moreover, the `Init()` function of this new c-class must first initialize the member `super` of the c-class and then initialize its other members:

```
1 XF_MODULE_DEFINE(XF<CLASSNAME>);
2 void XF<CLASSNAME>Init(XF<CLASSNAME>Init this, XFHashtable params)
3 {
4     /* initialize the base c-class */
5     XFModuleInit(&this->super, &gXF<CLASSNAME>Class);
6     /* initialize module specific data members */
7 }
```

Note that the `XFModuleInit()` function takes a `XFModuleClass` as parameter. The `XF_MODULE_DEFINE` macro automatically defines such a structure as static data with the name `gXF<Class>Class`.

Text processing module Process() function

The main function of a text processing module is its `XF<CLASSNAME>Process()` function. This function must return new data and its associated metadata. This function get the next data to process by calling the `Process()` function of the previous module. The module can also call the `XFModuleGetNextData()` function to get the next data to process. This function first check if there is some data left from the previous call that have not been processed. If no more data to process are available, it will call the `Process()` function of the previous module. Modules using `XFModuleGetNextData()` function must also call the `XFModuleClearPreviousData()` function to mark all previous data as “processed”

The `Process()` function must return `true` if new data are available.

It must also set the state of the module to `XFModuleStateEnd` when it has finished processing the last data from the previous module.

```

1 bool XF<CLASS>Process(XFModule super, XFUnicodeString result,
2                       XFString metadata)
3 {
4     XF<CLASS> this = XF<CLASS>super;
5     bool data_available = false;
6
7     if (XFModuleGetNextData(super)) {
8         data_available = true;
9         if (XFUnicodeStringLength(super->prev_result) > 0) {
10             /* Process new chunk... */
11         }
12         /* mark previous result as processed */
13         XFModuleClearPreviousData(super);
14     }
15     if (!data_available) {
16         XFModuleSetState(super, XFModuleStateEnd);
17     }
18
19     return data_available;

```

6 Modules available

This section contains the modules available in the XF toolkit. Their description contains the following information

- ➡ **Name:** name under which it is registered in the module manager.
- ➡ **Description:** purpose of the module.
- ➡ **Metadata:** metadata returned by the module
- ➡ **Parameters:** parameters that can or must be set when creating the module. Note that parameters without default value must be set. Other parameters are optional. The group is the parameter to specify to `setParameter()` function. Parameters belonging to the default group can be passed in the `CreateModule()` function. Others must be set by using the `setParameter()` method.

Modules are grouped into the following categories

- ➡ Input/Output modules
- ➡ Linguistic modules
- ➡ Utility modules

6.1 Input/Output modules

6.1.1 XFFileReader

Name

`filereader`

Description

The `filereader` module read its data from a file and convert them into Unicode

Metadata

`<file filename="[name-of-the-file]">`

Parameters

- ➡ Default group: none

Group	Name	Default value	Description
	<code>filename</code>		Name of the file to read
	<code>codeset</code>	<code>iso-8859-1</code>	Codeset in which the file is encoded.
	<code>bufsize</code>	<code>1024</code>	Number of bytes to read at each call.

6.1.2 XFStringReader

Name


`stringreader`

Description

The `stringreader` module read its data from a string and convert it into Unicode

Metadata

<string>

Parameters
 Default group: none

Group	Name	Default value	Description
	string		Text to process
	codeset	iso-8859-1	Codeset in which the string is encoded.

6.1.3 XFFileWriter**Name**


filewriter

Description

The `filewriter` module convert result to a specified codeset and write the converted string to a file. Metadata are ignored but passed to the next module. The result is passed unmodified to the next module.

Metadata

None.

Parameters
 Default group: none

Group	Name	Default value	Description
	filename		Name of the file to write
	codeset	utf-8	Codeset in which the file is encoded.

6.2 Linguistic modules**6.2.1 XFApplyFst****Name**

apply-fst

Description

The `apply-fst` module applies a finite-state transducer to an input string. The result of this module is the raw output of the apply function.

Metadata

None.

Parameters

🔑 Default group: fst

Group	Name	Default value	Description
fst	filename		Name of the transducer file
fst	default_dir	" "	Directory of the transducer file.

6.2.2 XFPriorityUnion

Name

priority-union

Description

The `priority-union` module applies a cascade of finite-state transducers to an input string until one cascade gives an output. A cascade simulate the composition of several transducers at run-time by apply the first transducer in the cascade to the input string and applying a transducer to the result of the previous transducer in the cascade. The result of the cascade is the output of the last transducer in the cascade.

Metadata

The following metadata are appended depending on the call number to this module:

Calls #	Metadata	Result
1	<surface_form>	Result of previous module
2	<senses-list><sense id='0'>	First result of the priority union
3	<senses-list><sense id='1'>	Second result of the priority union (if any)
4	etc	etc

Parameters

🔑 Default group: none

Group	Name	Default value	Description
	pos-tag	true	Specifies whether the cascades of transducers generates a morphological analysis with a part of speech tag or not.
	morpho-tags	false	Specifies whether the cascades of transducers generates a morphological analysis with all morphological tags or not.
	lookdown	false	Specifies the direction of the apply: lookup performs a morphological analysis and lookdown generates surface forms.
	process-result	true	Specifies whether the result of the lookup must be post processed by grouping morphological tags, and adding the confidence attribute to the part-of-speech tag.

Group	Name	Default value	Description
vectorize			Vectorize all transducers before using them to speed up processing.
vectorize	min-arcs	-1	The minimum number of output arcs for a state to be vectorized.
fst	filename		Name of the transducer file
fst	default_dir	" "	Directory of the transducer file.
fst	id		Identifier of this transducer used in the cascade definition
strategy	cascade		Blank separated list of fst id composing the cascade
strategy	merge-same-results	true	Set to false to avoid the merging of same result in the output. The merging is done in a very efficient way but add a little overhead to the processing.
script	filename		Name of the file containing the fst and strategy definitions in the same format as the one used by the lookup application.
script	default_dir	" "	Directory of the transducer file.
script	merge-same-results	true	Set to false to avoid the merging of same result in the output. The merging is done in a very efficient way but add a little overhead to the processing.

Notes:

- ➡ fst group parameters must be set before strategy group parameters
- ➡ It is not recommended to mix script and fst/cascade parameters.

6.2.3 XFDisambiguation

Name

disambiguation

Description

The `disambiguation` module disambiguates word analysis ambiguities based on their part of speech tags. Note that this module disambiguates only on the part of speech of the words and therefore keeps ambiguities on words having different analysis with the same part of speech.

Metadata

Keep metadata until the `<sentence>` tag is found. Replace the remaining metadata by

```
<sentence id='[sid] '><lexeme-list><lexeme id='[wid] '>
```

where

- ➡ `[sid]` is the sentence number computed from the end of sentence marker (word tagged +SENT) found in the text
- ➡ and `[wid]` is the word number in the sentence.

Then the following metadata are appended depending on the call number to this module:

Calls #	Metadata	Result
1	<surface_form>	Result of previous module
2	<senses-list><sense id='0'>	First result of the disambiguation
3	<senses-list><sense id='1'>	Second result of the disambiguation (if any)
4	etc	etc

Parameters

➡ Default group: hmm

Group	Name	Default value	Description
hmm	filename		Name of the file containing the HMM disambiguation matrices
hmm	default_dir	" "	Directory of the HMM matrices file.

6.3 Utility modules

6.3.1 XFSegmentation

Name

segmentation

Description

The `segmentation` module segments the input into group of tokens. The token separator can be any string.

Metadata

<[group]><[element] id='[id] '>

where

➡ [group] and [element] are the values of the group and element parameters.

➡ And [id] is the number of the token.

Parameters

➡ Default group: none

Group	Name	Default value	Description
	group		Tag to use for the group of elements. If not set, the <[group]> is not returned in the metadata
	element		Tag to use for each token.
	separator	\n	The string separator between each token.

6.3.2 XFFormatXML

Name

`format-xml`

Description

The `format-xml` module mixed the result and metadata to produce a XML compliant result.

Metadata

None

Parameters

🔑 Default group: none

Group	Name	Default value	Description
	<code>codeset</code>	<code>""</code>	The codeset in which the metadata of the previous module are encoded.

7 Using XF Low level API

The XF toolkit exposes a low-level API to the FST, tokenize and hmm libraries

7.1 XF FST low level API

7.1.1 XFFst

The `fst_utils.h` file groups a list of functions simplifying calls to the `fstbase` or `fstapply` libraries. It offers functions to init the libraries, load, free transducers and as well as performing some common operations.

For more information about these functions refer to the `XFFst` group in the XF Reference Manual.

Initialization, Cleanup

- ◆ `XFFstMemoryInit()`: Initialize the FST library.
- ◆ `XFFstMemoryFree()`: Free memory allocated by the FST library

Loading

- ◆ `XFFstLoadNetworks()`: Load a network and return FST internal structure

Internal symbol and strings

The FST library has its own way of representing symbols called “symbol id”, type `id_type`.

- ◆ `XFFstSymbolFromUnicodeString()`: Convert a character or multichar symbol encoded in a string to a FST symbol id.
- ◆ `XFFstSymbolToUnicodeString()`: Convert a FST symbol id to a printable unicode string.
- ◆ `XFFstSplitTags()`: splits a string into a vector of string where the first item is the first non multichar symbols (usually the baseform) and the next items are the multichar symbols (usually the tags).
- ◆ `XFFstIsMulticharSymbol()`: checks if a string is a multichar symbol.

A group of functions allow the manipulation of special Unicode String containing the FST internal symbols stored as characters in the Unicode “Private Use Area” `U+F0000..U+FFFFD`. Note that if the `UChar` type is 2 bytes long, such characters need 1 surrogate pair and therefore are stored in 2 `UChar` characters.

- ◆ `XFFstConvertPrivateSymbolsToString()`: Convert all characters of a string belonging the Unicode “Private Use Area” to an printable equivalent. For example the internal representation of the tag `+NOUN` is converted to the string `+NOUN`
- ◆ `XFFstAppendPrivateSymbolToUnicodeString()`: Convert a FST internal symbol to character in the “Private Use Area `U+F0000..U+FFFFD`” and append it the a string.
- ◆ `XFFstUCharToPrivateSymbolID()`: Get the FST symbol_id from a character belonging to the Unicode “Private Use Area”

- ➡ `XFFstPrivateSymbolIdToUChar()`: Convert a FST `symbol_id` to a Unicode character encoded in the private use area U+F0000..U+FFFFD. If the `UChar` type is 2 bytes long, then 1 surrogate pair (2 `UChar`) is necessary to encode the symbol id.
- ➡ `XFFstGetSymbolId()`: Convert a symbol string (i.e. "+NOUN") into the corresponding FST symbol id. If this symbol is not in the list of known symbols it is added. Warning: the symbol namespace is limited to 65535 symbols. Adding symbols has also a direct consequence on the memory space used by the vectorization.

Operations on collection of transducers

1 operation can be done on a collection of transducers:

- ➡ `XFFstVectorVectorizeAllNetworks()`: vectorizes all the transducers of a `XFFstVector` for speeding up the apply functions. Vectorizing a transducer consist in converting the list of out going arcs of a state into a vector of arcs indexed by the symbol id of the arc. This gives a direct access to the arc given an input symbol but increase the memory used by a transducer.
The `input_side` parameter is the side on which the string will be applied. It must be set to `LOWER` if this transducers will be used with a `Lookup()` function and `UPPER` if they will be used with `Lookdown()`.

7.1.2 XFFstVector

The `XFFstVector` c-class is a simple API to load, store and free transducers.

For more information about the functions of the `XFFstVector` c-class, refer to the XF Reference Manual.

Loading

Transducers can be loaded with the `XFFstVectorLoadFile()` and `XFFstVectorLoadFileList()` functions. They load the transducers stored in a file or space-separated list of files. Note that a `.fst` file can contains several transducers and therefore, even the `XFFstVectorLoadFile()` can load several transducers.

Freeing transducers

`XFFstVector` is a collection of transducers used by any function that need transducers as parameter. Therefore when deleting a `XFFstVector`, the transducer it contains are NOT automatically freed. To explicitly free them, call the `XFFstVectorFreeAll()` function.

Adding and getting transducers

For adding transducers to a `XFFstVector` use the `XFFstVectorAppendTransducers()` function. For getting the FST internal structure of a transducer, use the `XFFstVectorGetTransducers()` or `XFFstVectorGetTransducerAt()` functions.

7.1.3 XFFstApply

`XFFstApply` is a simple API to the `fst-apply` (aka `tokenize`) algorithm. Use the `XFFstApplyLoad()` function to load a transducer. Note `XFFstApplydelete()` function call the `XFFstVectorFreeAll()` function to free the transducer loaded.

Use `XFFstApplyLookup()` function to apply this transducer to a string. It returns a vector of Unicode string.

7.1.4 XFFstComposeApply

`XFFstComposeApply` is a simple API to the fst compose-apply algorithm. This algorithm can lookup/lookdown a string using a cascade of transducer. The results of the first transducer are used as input of the next one. The results (if any) are the output of the last one. It returns a vector of Unicode string.

Use the `XFFstComposeCascadeAppend()` function to append transducers to the cascade. Note that these transducers are NOT automatically freed when the `XFFstComposeApply` is deleted.

Use `XFFstComposeApplyLookup()` function to apply a string on the lower side of this cascade of transducers. It returns a vector of Unicode string. This is used typically to get all the possible morphological analysis of a word.

Use `XFFstComposeApplyLookdown()` function to apply a string on the upper side of this cascade of transducers. It returns a vector of Unicode string. This is used typically to generate all the possible words of a given morphological analysis.

7.1.5 XFFstStrategyApply

`XFFstStrategyApply` is a simple API to apply a series of `XFFstComposeApply` to an input string until one cascade gives an output.

A `XFFstStrategyApply` is composed of several cascades of transducers. For example, the first cascade can try to find an analysis of a word in the main lexicon as it is written whereas the second one will first normalize the word and find an analysis of a word in the main lexicon. To avoid having the same transducer loaded several times, such lexicon is shared by the 2 cascades.

The first step is to load the transducers that will be used in the different cascades and associate them a unique identifier (see `XFFstStrategyApplyLoadFst()`).

The second step is to create the cascade by giving the list of transducer id to use (see `XFFstStrategyApplyAddStrategy()`).

A convenience function can load the transducer and create the cascade from a "lookup script". (see `XFFstStrategyApplyLoadScript()`). The format of the lookup script is:

```
1 lex      lex+lemma.fst
2 norm     norm.fst
3 ...
4 <empty line>
5 lex
6 norm lex
7 ...
```

The file is composed of 2 parts separated with a blank line. The first part defines the transducers to load, the second one defines the cascades. In the previous example:

- The line 1 loads the `lex+lemma.fst` transducer and associates it the name "lex".
- The line 2 loads the `norm.fst` transducer and associates it the name "norm".
- The line 5 defines the first cascade composed only of the main lexicon

- ➡ The line 6 defines the second cascade composed of the normaliser followed by the main lexicon

Use `XFFstStrategyApplyLookup()` function to perform a morphological analysis of a word. It applies a string on the lower side of the cascades of transducers until a result is found. It returns a vector of Unicode string.

Use `XFFstStrategyApplyLookdown()` function to generate all possible words for a given morphological analysis. It applies a string on the upper side of the the cascades of transducers until a result is found. It returns a vector of Unicode string.

7.2 HMM low level API

XFHMM is a simple API to the part of speech disambiguation system using an HMM (Hidden Markov Model) algorithm.

Use `XFHMMLoad()` function to load the hmm model and `XFHMMDelete()` to free it.

The HMM system tries to disambiguate the words as soon as possible. Therefore results are available as soon as an unambiguous token is added to the list of word to disambiguate. The counterpart is that it must be informed when the end of the chunk of text has been reached to force the disambiguation of the remaining words in case the last token in the chunk was ambiguous.

Input

The XFHMM get input from:

- ➡ `XFHMMNewSentence()` : Must be called when a new chunk of text is starting.
- ➡ `XFHMMEndOfSentence()` : Must be called when the end of a chunk of text has been reached. It forces the remaining words to be disambiguated.
- ➡ `XFHMMNewLexeme()` : Must be called to start the disambiguation of a new word, along with its surface form.
- ➡ `XFHMMEndOfLexeme()` : Must be called when all the analysis of a word have been sent.
- ➡ `XFHMMNewAnalysis()` : Must be called for each morphological analysis of the current word.

Output

The output of this module is available through a `XFHMMResultIterator` returned by the `XFHMMGetResultIterator()` function when `XFHMMIsResultAvailable()` returns true.

Then the following functions are available:

- ➡ `XFHMMResultIteratorHasNextWord()` : returns true if there is at least one more word in the result iterator.
- ➡ `XFHMMResultIteratorNextWord()` : move iterator to next word.
- ➡ `XFHMMResultIteratorHasNextAnalysis()` : returns true if all the morphological analysis have not yet all been read.
- ➡ `XFHMMResultIteratorGetNextAnalysis()` : returns a vector containing the surface form as first item, the base form as second one followed by the tags. Note that

`XFHMMResultIteratorGetNextAnalysis()` **never** returns `NULL` if `XFHMMResultIteratorHasNextAnalysis()` **returned** `true`.

➡ `XFHMMResultIteratorIsEndOfSentence()` : **returns** `true` if the current word is a end of sentence marker.