X Window System, Version 11

Input Method Specifications

Public Review Draft - November 1990

(Send comments to i18n@expo.lcs.mit.edu)

Vania Joloboff
Open Software Foundation

Bill McMahon
Hewlett Packard Company

ABSTRACT

This chapter addresses the portability and interoperability of programs in different countries. It describes specifications providing to clients of the X Window System Version 11, an interface for input handling of characters in various languages. The specifications make it possible to develop portable applications independent of a particular language or a particular encoding of characters. The specifications are consistent with related specifications from X/Open Portability Guide, Release 3, and ANSI-C. The reader is assumed to be familiar with those, particularly with the notion of locale in the C language, therefore they will not be detailed here.

1. Input Method Overview

The next paragraphs provide definitions for terms and concepts used in the specification, and a brief overview of the intended use of the abstractions developed for Xlib internationalization.

1.1. What are Input Methods ?

A large number of languages in the world rely on an alphabet, a small set of symbols (letters) used to form words. To enter text into a computer in an alphabetic language a user usually has a keyboard on which there exists key symbols corresponding to the alphabet. Sometimes, a few characters of an alphabetic language are missing on the keyboard. Many computer users, who speak a Latin alphabet based language only have a English-based keyboard. They need to hit a combination of keystrokes in order to enter a character that does not exist directly on the keyboard. A number of algorithms have been developed for entering such characters, known as European input methods, or compose input method, or dead-keys input method.

In some alphabetic languages, the rendering of characters strings is context sensitive. When entering characters in those languages, a keystroke does not systematically mean appending a new symbol at the end of the string. It may modify the existing strings. Both input and output methods may be used in such languages.

With an ideographic writing system, rather than taking a small set of symbols and combining them in different ways to create words, each word consists of one unique symbol (or, occasionally, several symbols). The number of symbols may be very large: 150 000 have been identified in Hanzi, the Chinese ideographic system.

There are two major aspects of ideographics system for their computer usage. First, the standard computer character sets in Japan, China, and Korea include roughly 8 000 characters, while sets in Taiwan have between 15 000 and 30 000 characters, which make it necessary to use more than one byte to represent a character. Second, it obviously is impractical to have a keyboard that includes all of a given language's ideographic symbols. Therefore a specific mechanism is required for entering characters so that a keyboard with a reasonable number of keys can be used. Those input methods are usually based on the language's phonetics, but there also exist methods based on the graphics properties of characters.

In addition to the ideographic characters, a number of languages often also include a phonetic (alphabetic-based) writing system. The phonetic signs are then engraved on the keyboard and the keystrokes are transformed to their appropriate ideographic counterparts. Here's a brief description of the Japanese and Korean phonetic systems:

- **Japanese**: There are two phonetic symbol sets: *katakana* and *hiragana*. In general, you use katakana for words that are of foreign origin, and hiragana for writing native Japanese words. Collectively, the two systems are called *kana*. Each set consists of approximately 50 characters. You type either kana or English characters and define the region that you want to convert to kanji. Several kanji characters may have the same phonetic representation. If that's the case with your string, you get a menu of characters and choose the appropriate one. If no choice is necessary, the input method does the substitution directly. When Latin characters are converted to kana or Kanji, it is called a *romaji* conversion.
- **Korean:** *Hangul* is a writing system that actually straddles the line between phonetic and ideographic. It's phonetic in the sense that each of the roughly 25 characters represents a specific sound. But between two and five of the characters are combined to form syllables, and these syllables are the basic units on which text processing is done. For example, a delete operation works on a syllable rather than the individual characters within it. And Korean code sets include several thousands of these syllables. You type the hangul characters that make up the syllables of the words you're entering. The display changes as you enter each hangul letter. That is, when you enter the first letter, it fills the entire space that the final syllable will take up. When you enter the second, the first shrinks to about half its size to make room for the second. When you enter the third, the first two shrink again. And so on, up to the maximum of five letters in a syllable.

  It's usually acceptable to keep Korean text in hangul form, but some words are more commonly written in hanja. If you want to change hangul to hanja, you define the region to be converted, and follow the same basic method as described for Japanese.

Probably because there are well-accepted phonetic writing systems for Japanese and Korean, computer input methods for those languages are fairly standard. Keyboard keys have both English characters and the local language's phonetic symbols engraved on them. You can then switch the keyboard from English to local mode and vice versa.

The situation is different for Chinese. While there is a phonetic system called *Pinyin* promoted by authorities, there is no consensus for entering Chinese text. Some vendors use a phonetic decomposition (Pinyin or another), others use ideographic decomposition of Chinese words, with various implementations and keyboard layouts. There are about 16 known methods, none of which is a clear standard.

Also, there are actually two ideographic sets used: **Traditional Chinese**, (the original written Chinese) and **Simplified Chinese**. Several years back, the People's Republic Of China launched a campaign to simplify some ideographic characters and eliminate redundancies all together. Under the plan, characters would be streamlined every five years. Characters have been revised several times now, resulting in the smaller, simpler set that makes up **Simplified Chinese**.

1.1.1. Input Method Architecture

As shown in the previous paragraphs, there are many different input methods in use today, varying with language, culture, and history. A common feature of many input methods is that the user may type multiple keystrokes in order to compose a single character (or set of characters). The process of composing characters from keystrokes is called *pre-editing*. It may require complex algorithms and large dictionaries involving substantial computer resources.

Input methods may require one or more areas in which to show the feedback of the actual keystrokes, to propose disambiguation to the user, to list dictionaries, and so on. The input method areas with which we are concerned in this specification are as follows.

> The *Status* area is intended to be a logical extension of the LED's that exist on the physical keyboard. It is an output-only window which is intended to present the internal state of the input method that is critical to the user. The status area may consist of text data and bitmaps or some combination.
>
> The *PreEdit* area is intended to display the intermediate text for those languages that are composing prior to the client handling the data.
>
> The *Auxiliary* area is used for pop-up menus and customizing dialogs that may be required for an input method. There may be multiple Auxiliary areas for any input method. Auxiliary areas are managed by the input method independent of the client. Auxiliary areas are assumed to be a separate dialog which is maintained by the input method.

There are various user interaction styles used for pre-editing. The ones that this specification addresses are as follows.

> For *on-the-spot* input methods, pre-editing data will be displayed in the application window. Application data is moved to allow pre-edit data to be displayed at the point of insertion.
>
> *Over-the-spot* pre-editing means that the data is displayed in an input method window that is placed over the point of insertion.
>
> *Off-the-spot* pre-editing means that the pre-edit window is inside the client window, but not at the point of insertion. Often this type of window is placed at the bottom of the client window.
>
> *Root-window* pre-editing refers to input methods that use a pre-edit window that is the child of RootWindow.

It would require a lot of computing resources if portable applications had to include input methods for all the languages in the world. To avoid this, the goal of these specifications is to allow an application to communicate with an input method placed in a separate process. Such a process is called an *input server*. The server to which the application should connect is dependent upon the environment when the application is started up: what is the user language, the actual encoding to be used for it. We will say that input method connection is *locale dependent*. It is also user dependent: for a given language, the user can choose to some extent the user interface style of input method (if choice is possible among several).

Using an input server implies communication overhead, but applications can be migrated without relinking. Specifications in this document have been designed so input methods can be implemented either as a stub communicating to an input server or as a local library.

An input method may be based on a *front-end* or a *back-end* architecture. In front-end, there are two separate connections to the X server: keystrokes go directly from X server to the input method on one connection, other events to the regular client connection. The input method is then acting as a filter, and sends composed strings to the client. Front-end requires synchronization between the two connections to avoid lost key events or locking issues.

In back-end, a single X server connection is used. A dispatching mechanism must decide on this channel to delegate appropriate keystrokes to the input method. For instance, it may retain a *Help* keystroke for its own purpose. In the case where the input method is a separate process (i.e., a server), there must be a special communication protocol between the back-end client and the input server.

Front-end introduces synchronization issues and filtering mechanism for non-character keystrokes (Functions, Help, etc). Back-end sometimes implies more communication overhead and more process switching. If all three processes are running on a single workstation (X server, input server, client) there are two process switch for each keystroke in back-end, but only one in frontend.

The abstraction used by a client to communicate with an input method is an opaque data structure represented by the XIM data type. This data structure is returned by the **XOpenIM** function, which opens an input method on a given display. Subsequent operations on this data structure encapsulate all communication between client and input method. There is no need for an X client to use any networking library or natural language package in order to use an input method.

The specifications are not meant to provide explicit support to multi-lingual clients (using several languages), but they make it possible to implement such clients. For each language, a client has to open an input method for that language. Other functions will execute within the context of a given input method.

Characters from the various languages are represented in a computer using an encoding. Different languages have different encodings, and there are even different encodings for the same characters in the same language. A client program that can run using any encoding without any change in the source code is said to be *codeset independent*.

These specifications do not have specific requirements concerning languages or encodings supported by an input method or input server. A single input server may be used for one or more languages, supporting one or more encoding schemes. But the strings returned from an input method will always be encoded in the locale that was the current locale when the input method was opened.

1.2. Input Context

The specifications provide the ability to manage a multi-threaded state. A client may be using multiple windows, each window with multiple text entries, and the user possibly switching among them at any time. The abstraction for representing state of a particular input thread is called an *input context*. The Xlib representation of an input context is an XIC.

An input context is the abstraction retaining the state, properties and semantics of communication between a client and an input method. An input context is a combination of an input method, a locale specifying the encoding of the character strings to be returned, a client window, internal state information and various layout or appearance characteristics. The input context concept somewhat matches for input the graphics context abstraction defined for graphics output.

One input context belongs to exactly one input method. Different XIC's may be associated with the same input method, possibly with the same client window. An input context is created with the **XCreateIC** function, providing an XIM argument, affiliating the XIC to the input method for its lifetime. When an input method is closed with **XCloseIM**, all of its affiliated input context's should not be used any more (and preferably be destroyed too).

Considering the example of a client window with multiple text entries windows, the application programmer could for example choose to implement:

- As many input contexts are created as text entries and the client will get the input accumulated on each context each time it will lookup that context.
- A single context is created for a top level window in the application. If such window contains several text entries, each time the user moves to another text entry, the client has to indicate changes in the context.

A range of choices can be made by application designers to use either a single or multiple XIC's, according to the needs of their application.

1.3. Getting Input

In order to obtain characters from an input method a client must call the function **XmbLookupString** or **XwcLookupString** (noted as X*LookupString) with an XIC created from that XIM. Both a locale and display are bound to an input method when it is opened and an XIC inherits this locale and display. Any strings returned by X*LookupString will be encoded in that locale.

These functions are similar to XLookupString in X11 Release 4. One difference from XLookupString is that X*LookupString returns a Status. A programmer calling X*LookupString should check this status value to know if any valid input is being returned. If no input has been composed it will return a None status. >From the very nature of input methods, it may take multiple keystrokes before it returns a character or a string. To internationalize an X application, programmers should change the code so that each time it was receiving a key (press) event and called XLookupString, it would now call X*LookupString, and additionally check that something is returned and behave accordingly. Typically, a test should be inserted so that no action is taken if nothing is returned.

1.4. Focus management

For each text entry object which is doing input using either of the X*LookupString routines there will be an associated IC.

When a text entry object gets the application focus, it is the job of that text entry object to set the IC focus to the corresponding IC. The IC focus is set by calling **XSetICFocus** with the appropriate IC.

Also, when a text entry object looses the application focus, that text entry object should call **XUnSetICFocus**. As an optimization, if **XSetICFocus** is called successively on two different XIC's, setting the focus on the second IC will unset the focus on the first XIC.

Note that in order to set and unset the XIC focus correctly it will be necessary for a toolkit (or application) to track focus changes. Such focus changes do not necessarily correspond to X server focus changes.

If a toolkit (or application) is using a single XIC to do input for multiple text entry objects, it will also be necessary for that text entry object to set the focus window of the XIC whenever the focus window changes. (see **XNFocusWindow** under *XSetICValues*).

1.5. Geometry Management

In most input method architectures, (on-the-spot being the notable exception) the input method will perform the display of its own data. In order to provide better visual locality, it is often desirable to have the input method areas embedded within a client. In order to do this the client may need to allocate space for an input method. This spec provides support that allows the size and position of input method areas to be provided by a client. The input method areas that are supported for geometry management are the *Status* area and the *PreEdit* area.

The fundamental concept on which geometry management for input method windows is based is the proper division of responsibilities between the client (or toolkit) and the input method. The division of responsibilities is as follows.

- The client is responsible for the geometry of the input method window.
- The input method is responsible for the contents of the input method window.

An input method is able to suggest a size to the client, but it cannot suggest a placement. Also the input method can only suggest a size. It does not determine the size and it must accept the size it is given.

Before a client provides geometry management for an input method it must determine if geometry management is needed. The input method indicates the need for geometry management by setting XIMPreEditArea or XIMStatusArea in its XIMStyles value returned by **XGetIMValues**. When a client has decided that it will provide geometry management for an input method, it indicates that decision by setting the XNInputStyle value in the XIC.

After a client has established with the input method that it will will do geometry management, the client must negotiate the geometry with the input method. The geometry is negotiated by the following steps.

- The client suggests an area to the input method by setting the IC's AreaNeeded value for that area. If the client has no constraints for the input method it either will not suggest an area or will set the width and height to zero. Otherwise it will set one of the values.
- The client will get the XIC value AreaNeeded. The input method will return its suggested size in this value. The input method should pay attention to any constraints suggested by the client.
- The client sets the XIC value Area to inform the input method of the geometry of its window. The client should try to honor the geometry requested by the input method. The input method must accept this geometry.

Clients doing geometry management must be aware that setting other IC values may affect the geometry desired by an input method. For example, *FontSet* and *LineSpacing* may change the geometry desired by the the input method.

The table of XIC values in section 3 indicates the values that can cause the desired geometry to change when they are set. It is the responsibility of the client to re-negotiate the geometry of the input method window when it is needed.

In addition the specification provides a geometry management callback by which an input method can initiate a geometry change.

1.6. Filtering

A filtering mechanism is provided to allow input methods to capture X events transparently to clients. It is expected that toolkits (or clients) using X*LookupString will call this filter at some point in the event processing mechanism to make sure that events needed by an input method can be filtered by that input method.

If there were no filter, a client could receive and discard events that are necessary for the proper functioning of an input method. A few examples of such events are:

- Expose events on pre-edit window in local mode.
- Events may be used by an input method to communicate with an input server. Such input server protocol related events have to be intercepted if one does not want to disturb client code.
- Key events can be sent to a filter before they are bound to translations such as Xt provides.

Clients are expected to get the IC value XNFilterEvent and augment the event mask for the client window with that event mask. This mask may be NULL.

1.7. Callbacks

When an on-the-spot input method is implemented, only the client can insert/delete pre-edit data in place, and possibly scroll existing text. This means the echo of the keystrokes has to be achieved by the client itself, tightly coupled with the input method logic.

When a keystroke is entered, the client calls X*LookupString. At this point, in the on-the-spot case, the echo of the keystroke in the pre-edit has not yet been done. Before returning to the client logic that handles the input characters, X*LookupString must call the echoing logic for inserting the new keystroke. If the keystrokes entered so far make up a character, the keystrokes entered need to be deleted, and the composed character will be returned. Hence, what happens is that, while being called by client code, input method logic has to *call back* the client before it returns. As with the Xt toolkit, the client code, i.e a *callback* routine, is called from the input method logic.

There are a number of cases where the input method logic has to call-back the client. Each of those cases is associated with a well-defined callback action defined by these specifications. It is possible for the client to specify, for each input context, what callback is to be called for each action.

There are also callbacks provided for feedback of status information and a callback to initiate a geometry request for an input method.

2. Programming Interfaces.

Input method programming interface consists of a set of functions and data types starting with prefix X. The usual Xlib programming conventions apply to X function calls, including the two new resources XIM and XIC.

Most X interfaces have arguments which conform to ANSI-C variable argument list calling convention. Each procedure in the sequel denoted with a "..." argument takes a variable length list of name and value pairs where each name is of type XIMString and each value of type XIMValue. The end of the list is identified by a name argument containing NULL.

XIMValue is an implementation dependent typedef, which has to be large enough to contain void * in ANSI-C implementation, and any kind of pointer or long data in other C implementations.

```
typedef char * XIMString;

typedef void (*XIMProc)();

typedef SOMETHING XIMValue;
```

A variable length argument list may contain a nested list. If the name XNestedList is specified in place of an argument name, then the following value is interpreted as a **XVaNestedList** value which specifies a list of values logically inserted into the original list at the point of declaration. The end of a nested list is identified with a NULL name. To dynamically allocate a nested variable argument list, use **XCreateNestedList**.

```
#define XNVaNestedList "XNVaNestedList"

typedef void * XVaNestedList;
```

XVaNestedList XVaCreateNestedList(dummy, ...)
        int dummy; ANSI-C requirement for variable arguments.
        ...; specifies the variable length argument list.

XVaCreateNestedList allocates memory and copies its arguments into a single list pointer which may be used as value for arguments requiring a list value. Any entries are copied as specified. Data passed by reference is not copied; the caller must ensure data remains valid for the lifetime of the nested list. The list should be freed using XFree when no longer needed.

2.1. Opening, Closing, Querying Input Method

The functions to open and close a connection are **XOpenIM** and **XCloseIM**

XIM XOpenIM(*dpy*, *rdb*, *res_name*, *res_class*)
        Display **dpy*; specifies the display where the input method operates
        XrmDataBase *rdb*;
        XString *res_name*;
        XString *res_class*;

XOpenIM opens an input method, matching the current locale and modifiers specification. Current locale and modifiers are bound to the input method at opening time. The locale associated with an input method cannot be changed dynamically. This implies the strings returned by X*LookupString (see below), for any IC affiliated with a given input method, will be encoded in the locale current at the time input method is opened.

The specific input method to which this call will be routed is identified on the basis of the current locale. XOpenIM will identify a default input method corresponding to the current locale. That default can be modified using XSetModifier for the input method modifier.

Argument rdb is the resource database to be used by the input method for looking up resources that are private to the input method. It is not intended that this database be used to look up values that can be set as IC values in an input context. If rdb is NULL, no data base is passed to the input method.

Arguments res_name and res_class specify the resource name and class of the application. They are intended to be used as prefixes by the input method when looking up resources that are common to all input contexts that may be created for this input method. The characters used for resource names and classes must be in the X portable character set. The resources looked up are not fully specified if *res_name* or *res_class* is NULL.

XOpenIM returns NULL if no input method could be opened.

Status XCloseIM(im)
    XIM *im*;

XCloseIM closes the input method.

For querying input method use **XGetIMValues**.

void XGetIMValues(*im*, ...)
    XIM *im*; specifies the queried input method.
    ...; specifies the variable length argument list to query im values.

XGetIMValues presents a variable argument list programming interface for querying properties or features of the input method. Only one argument is defined by this version of the specifications: *XNQueryInputStyle*, that must be used to query about input styles supported by the input method.

A client should always query the input method to determine what styles are supported. The client should then find an input style it is capable of supporting.

If the client is not capable of meeting any of the requirements for input methods, it should ensure that the style XIMNeedNothing or XIMNeedNone is supported.

If the client cannot find an input style that it can support it should negotiate with the user the continuation of the program (exit, choose another input method, ...).

The argument value must be a pointer to a location where the returned value will be stored. The returned value is a pointer to a structure of type XIMStyles.

An XIMStyles structure contains in its field *count_styles* the number of input styles supported. This is also the size of the array in the field *supported_styles*.

Each element in the array represents a different input style supported by this input method. The value is of type XIMStyle. It is a bit mask in which the input method indicates its requirements, should this style be selected. These requirements fall into the following categories.

Preedit: this category defines what type of support is provided by the input method for pre-edit information:

XIMPreEditArea	If chosen, the input method would require the client to provide some area values for it to do its preediting. Refer to IC values XNArea and XNAreaNeeded.
XIMPreEditPosition	If chosen, the input method would require the client to provide positional values. Refer to IC values XNSpotLocation + XNFocusWindow.
XIMPreEditCallbacks	If chosen, the input method would require the client to define the set of preedit callbacks. Refer to IC values XNPreEditStartCallback, XNPreEditDoneCallback, XNPreEditDrawCallback, XNPreEditCaretCallback.
XIMPreEditNothing	If chosen, the input method can function without any PreEdit values.
XIMPreEditNone	The input method does not provide any PreEdit feedback. Any PreEdit value is ignored. This style is mutually exclusive with the other XIMPreEdit styles.

Status: this category defines what type of support is provided by the input method for status information:

XIMStatusArea	The input method requires the client to provide some area values for it to do its Status feedback. Refer to XNArea and XNAreaNeeded.
XIMStatusCallbacks	The input method requires the client to define the set of status callbacks; XNStatusStartCallback, XNStatusDoneCallback, XNStatusDrawCallback.
XIMStatusNothing	The input method can function without any Status values.
XIMStatusNone	The input method does not provide any Status feedback. If chosen, any Status value is ignored. This style is mutually exclusive with the other XIMStatus styles.

Clients are responsible for freeing the XIMStyles data structure, using XFree.

```
typedef unsigned short XIMStyle;

#define XIMPreEditArea          0x0001L
#define XIMPreEditCallbacks     0x0002L
#define XIMPreEditPosition      0x0004L
#define XIMPreEditNothing       0x0008L
#define XIMPreEditNone          0x0010L

#define XIMStatusArea           0x0100L
#define XIMStatusCallbacks      0x0200L
#define XIMStatusNothing        0x0400L
#define XIMStatusNone           0x0800L

typedef struct {
        unsigned short count_styles;
        XIMStyle * supported_styles;
        } XIMStyles;
```

3. Input Context

An input context is an abstraction used to contain the data required (if any) by an input method and the information required to display that data. There may be multiple input contexts for one input method. The programming interfaces for creating, reading or modifying an input context use a variable argument list. The name elements of the argument lists are referred to as IC values. It is intended that input methods be controlled by these IC values. As new IC values are created they should be registered with the X Consortium.

To create an input context use **XCreateIC**.

XIC XCreateIC(*im*, ...)
    XIM *im*; the attached input method.
    ...; specifies the variable length argument list to set slot values.

XCreateIC creates a context within an input method.

Some of the arguments are mandatory at creation time. The XIC will not be created if they are not provided. Those arguments are the input style and the set of text callbacks. (if the input style selected requires callbacks). All other input context values can be set later.

XCreateIC will return a NULL value if no XIC could be created. A NULL value could be returned for any of the following reasons: a required argument was not set, a read-only argument was set (e.g. XNFilterEvents), the argument name is not recognized, or the input method encountered an input method implementation dependent error.

The asynchronous errors that can be caused by **XCreateIC** are BadStyle, BadWindow, BadAtom, BadColormap, BadPixmap

The function **XDestroyIC** destroys an input context.

void XDestroyIC(*ic*)
    XIC *ic*;

ic must be a valid input context id.

Use functions **XSetICFocus** and **XUnSetICFocus** to communicate to and synchronize with input method for any changes in keyboard focus from the client side.

void XSetICFocus(*ic*)
    XIC *ic*; specifies the focussed XIC

XSetICFocus(ic) allows a client to notify an input method that the focus window attached to the ic argument has received keyboard focus. The input method should take action to provide appropriate feedback. Complete feedback specification is a matter of user interface policy, beyond the scope of these specifications.

void XUnSetICFocus(*ic*)
    XIC *ic*; specifies the focussed XIC

XICUnsetFocus allows a client to notify an input method that the ic has lost the keyboard focus and no more input is expected on the focus window attached to the specified ic. The input method should take action to provide appropriate feedback. Complete feedback specification is a matter of user interface policy, beyond the scope of these specifications.

To reset the state of an input context to initial state use either **XmbResetIC** or **XwcResetIC**.

char * XmbResetIC(*ic*)
    XIC *ic*;   specifies the XIC to be reset

XmbResetIC resets input context to initial state. Any input pending on that context is deleted. Input method is required to clear pre-edit area, if any, and update status accordingly. Calling XResetIC does not change the focus.

The return value of **XmbResetIC** is its current pre-edit string in multi-byte format. It is input method implementation dependent whether this routine ever returns any data.

The client should free this string using **XFree**.

char * XwcResetIC(*ic*)
    XIC *ic*;   specifies the XIC to be reset

XwcResetIC resets input context to initial state. Any input pending on that context is deleted. Input method is required to clear pre-edit area, if any, and update status accordingly. Calling XwcResetIC does not change the focus.

The return value of **XwcResetIC** is its current pre-edit string in wide-char format. It is input method implementation dependent whether this routine ever returns any data.

The client should free this string using **XFree**.

3.1. Utility functions

To get the XIM associated with an XIC, use **XIMOfIC.**

XIM XIMOfIC(ic)
    XIC *ic*;

To get the Display associated with an input method, use **XDisplayOfIM.**

Display * XDisplayOfIM(im)
    XIM *im*;

To get the locale associated with an input method, use **XLocaleOfIM.**

```
char * XLocaleOfIM(im)
        XIM im;
```

3.2. Modifying Input Contexts: XSetICValues and XGetICValues

Two functions allow for setting and reading XIC slots, respectively: **XSetICValues** and **XGetICValues**. Both functions have a variable length argument list. In that argument list, any slot name must be denoted with a character string using the X portable character set.

```
char * XSetICValues(ic, ...)
        XIC ic; specifies the input context to be changed
        ... ;  specifies the variable length argument list to get slot values.
```

Argument *ic* must be a valid input context id.

A value to be set must be an appropriate data, matching the data type imposed by the semantics of the argument.

Returns NULL if no error occurred, otherwise the name of the first argument that could not be set. An argument could be not set for any of the following reasons: a read-only argument was set (e.g. XNFilterEvents), the argument name is not recognized, or the input method encountered an input method implementation dependent error.

The asynchronous errors that can be caused by **XSetICValues** are BadWindow, BadAtom, BadColormap, BadPixmap

```
char * XGetICValues(ic, ...)
        XIC ic;
        ...; specifies the variable length argument list to get slot values.
```

Argument ic must be a valid input context id.

Any argument value in a variable argument list (following a name) must point to a location where the value is to be stored. **XGetICValues** will allocate memory to store the values, and client is responsible for freeing it, using **XFree.**

Returns NULL if no error occurred, otherwise the name of the first argument that could not be obtained. An argument could be not obtained for any of the following reasons: the argument name is not recognized, or the input method encountered an input method implementation dependent error.

3.3. IC Values Arguments

The following tables describe how IC values are interpreted by an input method depending on the input style chosen by the user.

The first column lists the IC values. The second column indicates which values are involved in affecting, negotiating and setting the geometry of the input method windows. The sub-entries under the third column indicate the different input styles supported. Each of these columns indicates how each of the IC values are treated by that input style.

IC Value	Geometry Management	Input Style				
		PreEdit Callback	PreEdit Position	PreEdit Area	PreEdit Nothing	PreEdit None
Input Style	GN	C-G	C-G	C-G	C-G	C-G
Client Window		O-G	O-G	O-G	ignored	ignored
Focus Window		D-S-G	D-S-G	D-S-G	D-S-G	ignored
Resource Name		ignored	D-S-G	D-S-G	D-S-G	ignored
Resource Class		ignored	D-S-G	D-S-G	D-S-G	ignored
Geometry Callback		ignored	ignored	D-S-G	ignored	ignored
FilterEvents		G	G	G	G	ignored
PreEdit						
- Area	GS	ignored	D-S-G	D-S-G	ignored	ignored
- AreaNeeded	GN-GR	ignored	ignored	S-G	ignored	ignored
- SpotLocation		ignored	C-S-G	ignored	ignored	ignored
- Colormap		ignored	D-S-G	D-S-G	D-S-G	ignored
- Fg/Bg		ignored	D-S-G	D-S-G	D-S-G	ignored
- Bg Pixmap		ignored	D-S-G	D-S-G	D-S-G	ignored
- FontSet	GN	ignored	C-S-G	C-S-G	D-S-G	ignored
- LineSpacing	GN	ignored	D-S-G	D-S-G	D-S-G	ignored
- Cursor		ignored	D-S-G	D-S-G	D-S-G	ignored
- PreEdit Callbacks		C-S-G	ignored	ignored	ignored	ignored

IC Value	Geometry Management	Input Style			
		Status Callback	Status Area	Status Nothing	Status None
Input Style	GN	C-G	C-G	C-G	C-G
Client Window		O-G	O-G	O-G	ignored
Focus Window		D-S-G	D-S-G	D-S-G	ignored
Resource Name		ignored	D-S-G	D-S-G	ignored
Resource Class		ignored	D-S-G	D-S-G	ignored
Geometry Callback		ignored	D-S-G	ignored	ignored
FilterEvents		G	G	G	G
Status					
- Area	GS	ignored	D-S-G	ignored	ignored
- AreaNeeded	GN-GR	ignored	S-G	ignored	ignored
- Colormap		ignored	D-S-G	D-S-G	ignored
- Fg/Bg		ignored	D-S-G	D-S-G	ignored
- Bg Pixmap		ignored	D-S-G	D-S-G	ignored
- FontSet	GN	ignored	C-S-G	D-S-G	ignored
- LineSpacing	GN	ignored	D-S-G	D-S-G	ignored
- Cursor		ignored	D-S-G	D-S-G	ignored
- Status Callbacks		C-S-G	ignored	ignored	ignored

Key:

C	This value must be set with **XCreateIC()**.
D	This value may be set using **XCreateIC()**. If not set, a default is provided.
G	This value may be read using **XGetICValues()**.
GN	This IC value may cause geometry negotiation when its value is set via **XCreateIC()** or **XSetICValues()**.
GR	This value will be the response of the input method when any *GN* value is changed.

GS This value will cause the geometry of the input method window to be set.

O This value must be set once and only once. It need not be set at IC create time.

S This value may be set with **XSetICValues()**.

ignored
The IC value is ignored by the input method for the given input style.

3.3.1. Input Style

Argument **XNInputStyle** specifies the input style to be used. The value of this argument must be one of the values returned by function XGetIMValues with argument XNQueryInputStyle in the supported_styles list, otherwise error BadStyle will be raised.

This argument must be set at creation time and cannot be changed.

3.3.2. Client Window

Argument **XNClientWindow** specifies to input method the client window in which the input method can display data or create subwindows. Geometry values for input method areas are given with respect to the client window. Dynamic change of client window is not supported. This argument may be set only once. This argument should be set before any input is done using this input context. If it is not set the input method may not operate correctly.

If an attempt is made to set this value a second time with **XSetICValues**, the string "XNClientWindow" will be returned by **XSetICValues** and the client window shall not be changed.

If the client window is not a valid window id on the display attached to the input method this argument can generate a BadWindow error when this value is used by the input method.

3.3.3. Focus Window

Argument **XNFocusWindow** specifies the focus window. The primary purpose of the **XNFocusWindow** is to identify the window which will receive the key event when input is composed. In addition, the input method may possibly affect the focus window: select events on it, send events to it, modify its properties, grab keyboard within that window.

The value associated to the argument must be of type Window. If the focus window is not a valid window id on the display attached to the input method this argument can generate a BadWindow error when this value is used by the input method.

When this slot is left unspecified, input method will default focus window to client window.

3.3.4. Resource Name and Class

Arguments **XNResourceName** and **XNResourceClass** are strings that represent the full name and class hierarchy used by the client to obtain resources for the client window. These values should be used as prefixes for name and class when looking up resources that may vary according to the IC. If these values are not set, the resources will not be fully specified.

It is not intended that values which can be set as IC values be set as resources.

3.3.5. Geometry Callback

The argument **XNGeometryCallback** is a structure of type XIMCallback (see PreEdit/Status Callbacks).

The argument **XNGeometryCallback** specifies the geometry callback which a client can set. This callback is not required for correct operation of either an input method or a client. It can be set for a client whose user interface policy permits an input method to request the dynamic change of that input methods window. An input method that does dynamic change will need to filter any events that it uses to initiate the change.

3.3.6. Filter Events

The argument **XNFilterEvents** will return the event mask that an input methods needs to have selected for. The client is expected to augment its own event mask for the client window with this one.

This argument is read-only. It is set by the input method at create time and is never changed.

The type of this argument is unsigned long. Setting this value will cause an error.

3.3.7. *PreEdit* and *Status* Attributes

Arguments **XNPreeditAttributes** and **XNStatusAttributes** specify to input method the attributes to be used for the *PreEdit* and *Status* areas, if any. Those attributes are passed to XSetICValues or XGetICValues as a nested variable length list. The names to be used in such lists are as described in the next subsection.

3.3.7.1. Area

The value of the argument **XNArea** must be a pointer to a structure of type XRectangle. The interpretation of argument **XNArea** is dependent on the input method style set in the IC.

If the input method style is **XIMPreEditPosition**, **XNArea** specifies the clipping region within which preediting will take place. If the focus window has been set the coordinates are assumed to be relative to the focus window. If the focus window has not been set the coordinates are assumed to be relative to the client window. If neither has been set, the results are undefined. If **XNArea** is not specified the input method will default the clipping region to the geometry of the **XNFocusWindow**. If the area specified is NULL or invalid the results are undefined.

If the input style is **XIMPreEditArea** or **XIMStatusArea, XNArea** specifies the geometry provided by the client to the input method. The input method may use this area to display its data, either *PreEdit* or *Status* depending on the area designated. The input method may create a window as a child of the client window with dimensions that fit the **XNArea**. The coordinates are relative to the client window. If the client window has not been set yet, the input method should save these values and apply them when the client window is set. If **XNArea** is not specified, is set to NULL or is invalid, the results are undefined.

3.3.7.2. Area Needed

When set, argument **XNAreaNeeded** specifies the geometry suggested by the client for this area (*PreEdit* or *Status*). The value associated with the argument must be a pointer to a structure of type XRectangle. The x, y values are not used. Non-zero values for width or height are constraints that the client wishes the input method to respect.

When read, argument **XNAreaNeeded** specifies the preferred geometry desired by the input method for the area.

This argument is only valid if the input style is **XIMPreEditArea** or **XIMStatusArea**. It is used for geometry negotiation between the client and the input method and has no other effect upon the input method (refer to Geometry Management.)

3.3.7.3. Spot Location

Argument **XNSpotLocation** specifies to input method the coordinates of the "spot", to be used by an input method executing with **XNInputStyle** set to **XIMPreEditPosition**. When specified to any input method other than **NeedPosition** this slot is ignored.

The *x* coordinate specifies the position where the next character would be inserted. The *y* coordinate is the position of the baseline used by current text line in the focus window. The *x* and *y* coordinates are relative to the focus window if it has been set, else they are relative to the client window. If neither focus window nor client window has been set the results are undefined.

The value of the argument is a pointer to a structure of type XPoint.

3.3.7.4. Colormap

Two arguments can be used to indicate what colormap the input method should use to allocate colors: one as a colormap XID, the other as a standard colormap name. Argument **XNColormap** is used to specify a colormap id. Argument value is of type **Colormap**. This argument may generate a BadColormap error when the colormap is used by the input method. Argument **XNStdColormap** is used to indicate the name of the standard colormap in which input method should to allocate colors. Argument value is an Atom that should be a valid atom for calling XGetStandardColormap. This argument may generate BadAtom or BadColormap error when it is used by the input method.

If colormap is left unspecified, it is defaulted to client window colormap.

3.3.7.5. Foreground and background.

Arguments **XNForeground** and **XNBackground** specifies respectively the foreground and background pixel. Argument value is of type *unsigned long*. It must be a valid pixel in the input method colormap. Error BadPixel is raised when this argument is used by the input method if pixel is not a valid pixel for the colormap in use by the input method.

If these values are left unspecified, the default is determined by the input method.

3.3.7.6. Background pixmap.

Argument **XNBackgroundPixmap** specifies a background pixmap to be used as the background of the window. Slot value must be of type *Pixmap*. Can generate error BadPixmap when this argument is used by the input method.

If this value is left unspecified, the default is determined by the input method.

3.3.7.7. FontSet

Argument **XNFontSet** specifies to input method what fontset is to be used. Argument value is of type XFontSet.

If this value is left unspecified, the default is determined by the input method.

3.3.7.8. Line Spacing

Argument **XNLineSpace** specifies to input method what line spacing is to be used in pre-edit window if more than one line is to be used. The slot value is of type *int*.

If this value is left unspecified, the default is determined by the input method.

3.3.7.9. Cursor

Argument **XNCursor** specifies to input method what cursor is to be used in the specified window. The slot value is of type *Cursor*.

If this value is left unspecified, the default is determined by the input method.

3.3.7.10. PreEdit/Status Callbacks

A client that wishes to support the input style XIMPreEditCallbacks must provide a set of pre-edit callbacks to the input method. The set of pre-edit callbacks are:

- **XNPreEditStartCallback** called when the input method starts pre-edit.
- **XNPreEditDoneCallback** called when the input method stops pre-edit.
- **XNPreEditDrawCallback** called when a number pre-edit keystrokes should be echoed.
- **XNPreEditCaretCallback** called to move text insertion point within pre-edit string

A client that wishes to support the input style XIMStatusCallbacks must provide a set of status callbacks to the input method. The set of status callbacks are:

- **XNStatusStartCallback** called when the input method initializes status area.
- **XNStatusDoneCallback** called when the input method no longer needs status area.
- **XNStatusDrawCallback** called when updating the status area is required.

The value of any status or pre-edit argument is a pointer to a structure of type XIMCallback.

```
typedef struct {
        XIMValue client_data;
        XIMProc callback;
        } XIMCallback;
```

Each callback has some particular semantics and will carry the data expressing the environment necessary to the client into a specific data structure. This paragraph only describes the arguments to be used to set the callback. For a complete description of the semantics see section 4.

Setting any of these values while doing preedit may cause unexpected results.

4. Callbacks semantics

Callbacks are functions defined by clients or text drawing packages, to be called from the input method when selected events occur. Most clients will use a text editing package, or a toolkit, and hence will not need to define such callbacks. This section defines the callback semantics, when they are triggered, and what their arguments are; it is mostly useful for toolkit implementors.

Callbacks are mostly provided so that clients (or text editing packages), can implement on-the-spot pre-editing in their own window. In that case, the input method needs to communicate and synchronize with the client. Input method needs to communicate changes in the pre-edit window when it is under control of the client. Those callbacks allow the client to initialize the pre-edit area, display a new pre-edit string, move the text insertion point inside pre-edit, terminate pre-edit, update the status area.

All callback functions follow the generic prototype:

void CallbackPrototype(*ic*, *client_data*, *call_data*)
    XIC *ic*; specifies the originator of callback.
    XIMValue *client_data*; specifies additional client data.
    XCallbackStruct **callback_data*; specifies specific callback data.

The callback_data is a structure expressing the arguments needed to achieve the semantics: a specific data structure appropriate to the callback. In cases where no data is needed in the callback, this callback_data is NULL. The client_data argument is a closure, initially specified by the client when specifying the callback and passed back. It may serve, for example, to inherit application context in the callback.

The following paragraphs describe the semantics and specific data structure associated with the different reasons.

4.1. Geometry Callback

The geometry callback is triggered by the input method to indicate that it wants the client to negotiate geometry.

void GeometryCallback(*ic*, *client_data*, *call_data*)
    XIC *ic*; specifies the originator of callback.
    XIMValue *client_data*; client data.
    void * *call_data*; specific callback data.

GeometryCallback is called with a NULL call_data argument.

4.2. PreEdit State Callbacks

When the input method turns input conversion on or off, PreEditStartCallback or PreEditDoneCallback is triggered in order to let the toolkit do the setup or the cleanup for the pre-edit region.

int PreEditStartCallback(*ic*, *client_data*, *call_data*)
XIC *ic*; specifies the originator of callback.
XIMValue *client_data*; client data.
void * *call_data*; specific callback data.

When pre-edit starts on the specified ic, the callback is called, with a NULL call_data argument. **PreEditStartCallback** will return the maximum size of the pre-edit string. A positive number indicates the maximum number of bytes allowed in the pre-edit string, a value of -1 indicates there is no limit.

void PreEditDoneCallback(*ic*, *client_data*, *call_data*)
XIC *ic*; specifies the originator of callback.
XIMValue *client_data*; client data.
void * *call_data*; specific callback data.

When pre-edit stops on the specified ic, the callback is called, with a NULL call_data argument. The client can release the data allocated by PreEditStartCallback.

PreEditStartCallback should initialize appropriate data needed for displaying pre-edit information and for handling further PreEditDrawCallback calls. Once PreEditStartCallback is called, it shall not be called again before PreEditDoneCallback has been called.

4.3. PreEditDraw Callback

This callback is triggered to draw and insert, delete or replace, pre-edit text in the pre-edit region. The pre-edit text may include unconverted input text such as Japanese kana, converted text such as Japanese Kanji characters, or characters of both kinds. That string is either a multi-byte or wide-character string, whose encoding matches the locale bound to the XIC. The callback prototype is as follows:

void PreEditDrawCallback(*ic*, *client_data*, *call_data*)
XIC *ic*; the originator of callback.
XIMValue *client_data*; client data.
XIMPreEditDrawCallbackStruct **call_data*; callback data.

The callback is passed a XIMPreEditDrawCallbackStruct structure in the call_data argument. The text member of this structure contains the text to be drawn. After the string has been drawn, the caret should be moved to the specified location.

```
typedef struct _XIMPreEditDrawCallbackStruct {
        int caret;      /* Cursor offset within pre-edit string */
        int chg_first;          /* Starting change position */
        int chg_length;     /* Length of the change in character count */
        XIMText text;
        } XIMPreEditDrawCallbackStruct ;
```

The client must keep updating a buffer of the pre-edit text, the callback arguments referring to indexes in that buffer. The call_data fields have specific meanings according to the operation:

- To indicate text deletion, the call_data specifies a NULL text field. The text to be deleted is then the current text in buffer from position chg_first (starting at zero) on a (character) length of chg_length.
- When text is non-NULL it indicates insertion or replacement of text in the buffer.

  A positive chg_length indicates that the characters starting from chg_first to ch_first+chg_length must be deleted, and replaced by text, whose length is specified in the XIMText structure.

  A chg_length value of 0 indicates that text must be inserted right at the position specified by chg_first. A value of 0 for chg_first specifies the first character in the buffer.
- caret: index in the the pre-edit text buffer specifying the character after which the cursor should move after text has been drawn or deleted.

```
typedef struct _XIMText {
     unsigned short length;
```

```
XIMFeedback * feedback;
    Bool encoding_is_wchar;
    union {
    char * multi_byte;
    wchar_t * wide_char;
    } string;
    } XIMText;
```

The text string passed is actually a structure specifying:

- length: the text length in characters.
- encoding_is_wchar: indicates if the string is passed encoded in wide character or multi-byte.
- string: the text string.
- feedback: indicates rendering type.

The feedback field express the types of rendering feedback the callback should apply when drawing text. Rendering of the text to be drawn is specified either in generic ways (e.g. primary, secondary) or in specific ways (reverse, underline). When generic indications are given, the client is free to choose the rendering style. It is necessary however that primary and secondary are mapped to two distinct rendering styles.

The feedback field specifies how the rendering of the text argument should be achieved. If feedback is NULL, then rendering is assumed to be the same as rendering of other characters in the text entry. Otherwise, feedback specifies an array defining the rendering of each character of the string (hence the length of the array is length).

If an IM wishes to indicate that it is only updating the feedback of the pre-edit text without changing the content of it, XIMText should contain a NULL value for the string field, the number of characters affected in the length field and the feedback field should point to an array of XIMFeedback.

Each element in the array is a bit mask represented by a value of type XIMFeedback. The valid masks names are as follows.

```
typedef unsigned char XIMFeedback;

#define     XIMReverse      1L
#define     XIMUnderline    (1L<<1)
#define     XIMHighlight    (1L<<2)
#define     XIMPrimary      (1L<<6)
#define     XIMSecondary    (1L<<7)
#define     XIMTertiary     (1L<<8)
```

4.4. PreEditCaretCallback

An input method may have its own "navigation keys" to allow the user to move the text insertion point in the pre-edit area (e.g. move backward or forward). Consequently, input method needs to indicate to the client that it should move the text insertion point. It then calls the **PreEditCaretCallback**

void PreEditCaretCallback(*ic*, *client_data*, *call_data*)
XIC *ic*; the originator of callback.
XIMValue *client_data*; specifies additional client data.
XPreEditCaretCallbackStruct **call_data*; specific callback data.

Input method will trigger PreEditCaretCallback to move the text insertion point during pre-edit. The call_data argument contains a pointer to an XPreEditCaretCallback structure. This structure indicates where the caret should be moved. The callback must move the insertion point to its new location and return, in field position, the new offset value from initial position.

```
typedef struct _XIMPreEditCaretCallbackStruct {
        int position; /* Caret offset within pre-edit string */
```

```
     XIMCaretDirection direction;       /* Caret moves direction */
        XIMCaretStyle style;            /* Feedback of the caret */
        } XIMPreEditCaretCallbackStruct ;

typedef enum {
       XIMInvisible,        /* Disable caret feedback */
       XIMPrimary,/* UI defined caret feedback */
       XIMSecondary,        /* UI defined caret feedback */
} XIMCaretStyle;

typedef enum {
       XIMForwardChar, XIMBackwardChar,
       XIMForwardWord, XIMBackwardWord,
       XIMCaretUp, XIMCaretDown,
        XIMNextLine, XIMPreviousLine,
       XIMLineStart, XIMLineEnd,
       XIMAbsolutePosition,
        XIMDontChange
 } XIMCaretDirection;
```

The meaning of those values are:

- XIMForwardChar: Move caret forward one character position
- XIMBackwardChar: Move caret backward one character position
- XIMForwardWord: Move caret forward one word position
- XIMBackwardWord: Move caret backward one word position
- XIMCaretUp: Move caret up one line keeping current offset.
- XIMCaretDown: Move caret down one line keeping current offset.
- XIMPreviousLine: Move caret up one line.
- XIMNextLine: Move caret down one line
- XIMLineStart: Move caret to beginning of the current display line containing the caret.
- XIMLineEnd: Move caret to end of the current display line containing the caret.
- XIMAbsolutePosition: The callback must move to the location specified by the position field of the callback data, indicated in characters, starting from the beginning of the pre-edit text. Hence a value of zero means: move back to beginning of the pre-edit text.
- XIMDontChange: The caret position does not change.

4.5. Status Callbacks

An input method may communicate changes in the status of an input context (created, destroyed, or focus changes), with three status callbacks: **StatusStartCallback** , **StatusDoneCallback** , **StatusDrawCallback**

When the XIC is created or when the XIC gains focus, IM calls **StatusStartCallback** callback.

void StatusStartCallback(*ic*, *client_data*, *call_data*)
XIC *ic*; the originator of callback.
XIMValue *client_data*; specifies additional client data.
XIMValue *call_data*; specific callback data.

The callback should initialize appropriate data for displaying status and be prepared to further StatusDrawCallback calls. Once StatusStartCallback is called, it shall not be called again before StatusDoneCallback has been called.

When an XIC is destroyed or when focus is lost by the XIC, IM calls **StatusDoneCallback.**

void StatusDoneCallback(*ic*, *client_data*, *call_data*)
XIC *ic*; the originator of callback.
XIMValue *client_data*; specifies additional client data.
XIMValue *call_data*; specific callback data.

The callback may release any data allocated on StatusStart.

When an XIC status has to be updated, IM calls **StatusDrawCallback.**

void StatusDrawCallback(*ic*, *client_data*, *call_data*)
XIC *ic*; the originator of callback.
XIMValue *client_data*; specifies additional client data.
XIMValue *call_data*; specific callback data.

The callback should update the status area by either drawing a string, or imaging a bitmap in the status area.

```
typedef enum {
          TextType,
          Bitmaptype,
     } XIMStatusDataType;

typedef struct _XIMStatusDrawCallbackStruct {
     XIMStatusDataType      type;
     union         {
               XIMText text;
               Pixmap  bitmap;
          } data;
     } XIMStatusDrawCallbackStruct ;
```

5. Filtering

Xlib provides the ability for an input method (or any other code) to register a filter with Xlib. This filter is called by a client (or toolkit) by calling **XFilterEvent** after calling XNextEvent. Any client that uses the XIM interface should call XFilterEvent to allow input methods to process their events without knowledge of the client's dispatching mechanism. A client's user interface policy may determine the priority of event filters with respect to other event handling mechanisms, (e.g. modal grabs).

A package that wants to do event filtering, e.g. an input method, must register its filter with the filtering machinery. It can add or remove filters using the two functions **XRegisterFilter**, **XUnRegisterFilter** There may be zero or more filters for the same event. Clients may not know how many filters there are, if any, and what they do. They may only know if an event has been filtered on return of XFilterEvent. Clients should discard filtered events.

Boolean XFilterEvent(*evt*, *window*)
XEvent * *evt*;
Window *window*; specifies the window for which this filter should be applied

XFilterEvent is called with an XEvent and a Window. *Window* specifies the window for which this filter should be applied. If *window* is NULL, the window from *evt* is applied. *Window* is provided so that layers above Xlib that do event redirection can indicate to which window an event has been redirected.

XFilterEvent searches its list of filters for a filter that matches the display and event type from the event and *window*. When XFilterEvent finds a filter that matches it will call that filter. If that filter returns True, XFilterEvent immediately returns True, without looking for more filters. If no filters are found that filter the event, False is returned. Filters are called in the order in which they were registered.

If a grab has occurred in the client, and the filter is returning True, the client should ungrab the keyboard.

To register a filter, use **XRegisterFilter**

void XRegisterFilter(*display*, *window*, *event_mask*, *filter*, *client_data*)
Display * *display*; specifies the display for which this filter is being registered
Window *window*; specifies the window for which this filter is being registered
unsigned long *event_mask*; specifies the event mask for which to call the filter.
Boolean *nonmaskable*; specifies whether the filter should be called on the nonmaskable events
Boolean *(*filter)()*; specifies the filter to be called
XIMValue *client_data*; additional client data to be passed to the filter.

XRegisterFilter registers a filter with the filtering machinery. This filter will be called on further calls to XFilterEvent for events generated on the specified display and window, when the event matches the mask. If *window* is NULL, the filter will match for any window value.

For each set of *display*, *window*, *filter*, *client_data*, exactly one filter will be registered. If a filter is registered that matches a currently registered filter, then the specified *event_mask* augments the existing mask. Clients who want to filter nonmaskable events (GraphicsExpose, NoExpose, SelectionClear, SelectionNotify, SelectionRequest, ClientMessage and MappingNotify) should set nonmaskable to True.

XUnRegisterFilter(*display*, *window*, *filter*, *client_data*)
Display * *display*; specifies the display for which this filter is being removed.
Window *window*; specifies the window for which this filter is being removed.
Boolean *(*filter)()*; specifies the filter that is to be removed
XIMValue *client_data*; specifies client data registered

XUnRegisterFilter removes a filter registered with **XRegisterFilter**. If *display*, *window*, *filter* and *client_data* do not match a registered filter, no filter is removed and no error is reported.

The filter prototype is as follows.

Boolean PrototypeFilter(*display*, *window*, *evt*, *f_data*)
Display * *display*; the display for which filter is being called.
Window *window*; the window for which filter is being called.
XEvent **evt*; the triggering event
XIMValue *client_data*; client data previously registered
{
/* if event is filtered return True, else False */
}

A filter function should not remove events from the queue (e.g. call XNextEvent). It should decide from context if the event is filtered or not.

6. Getting Composed Input

To get composed input from an input method, use **XmbLookupString** or **XwcLookupString**.

int XmbLookupString(*ic*, *event*, *buffer_return*, *bytes_buffer*, *keysym_return*, *status_return*)
XIC *ic*; specifies context for input processing.
XKeyPressedEvent **event*; specifies key press event.
char **buffer_return*; specifies multibyte string return buffer.
int *bytes_buffer*; specifies space available in return buffer.
KeySym **keysym_return*; specifies Out: return keysym (if any).
Status **status_return*; flag specifying what is returned.

int XwcLookupString(*ic*, *event*, *buffer_return*, *bytes_buffer*, *keysym_return*, *status_return*)
XIC *ic*; specifies context for input processing.
XKeyPressedEvent **event*; specifies key press event.
wchar_t **buffer_return*; specifies wide char return buffer.
int *wchars_buffer*; specifies space available in buffer.
KeySym **keysym_return*; specifies returned keysym (if any).
Status **status_return*; flag specifying what is returned.

These functions return the string from the input method in *buffer_return*. If no string is returned *buffer_return* is unchanged.

The keysym into which the keycode from the event was mapped is returned in *keysym_return* if *keysym_return* is non-NULL and *status* indicates that a KeySym was returned. If both a string and a keysym are returned, the keysym value does not necessarily correspond to the string returned.

XmbLookupString will return the length of the string in bytes. **XwcLookupString** will return the length of the string in characters.

These functions also detect any rebound keysyms (see XRebindKeysym) and return the specified bytes. since the last call.

The *ic* argument specifies an input context identifier returned from the XCreateIC() call. Unlike XLookupString, which returns only Latin-1 strings, XmbLookupString and XwcLookupString return text in the codeset of the locale bound to the input method of the specified input context.

Note that each string returned by Xmb/wcLookupString begins in the initial state of the codeset of the locale (if the codeset of the locale is state-dependent).

Caution: in order to insure proper input processing, it is essential that the client pass only KeyPress events to X*LookupString. The behavior of X*LookupString when the client passes a KeyRelease event is undefined.

Clients should check *return_status* before using the other returned values. These two functions both return in *return_status* a value indicating what has been returned in the other arguments:

- **XBufferOverflow** means that the input string to be returned is too large for the supplied buffer_return. The required size (XmbLookupString in bytes, XwcLookupString in characters) is returned as the value of the function, and the contents of buffer_return and keysym_return are not modified. The client should recall the function with the same event and a buffer of adequate size in order to obtain the string.
- **XLookupNone** means that no consistent input has been composed so far. The contents of buffer_return and keysym_return are not modified, and the function returns 0 as a value.
- **XLookupChars** means some input characters have been composed. They are placed in *buffer_return* and the string length in octets is returned as the value of the function. The string is encoded in the locale bound to the input context. The contents of keysym_return is not modified.
- **XLookupKeySym** means a KeySym has been returned instead of a string. The keysym is returned in keysym_return. The contents of *buffer_return* is not modified, and the function returns 0 as a value.
- **XLookupBoth** means that both a KeySym and a string are returned, XKeySym and XString occur simultaneously.

It does not make any difference if the input context passed as an argument to X*LookupString is the XIC currently in possession of the focus or not. Input may have been composed within an input context before it lost the focus and that input may be returned on subsequent calls to X*LookupString even though it does not have any more keyboard focus.

7. Conventions

The various input method architecture are transparent to the client. However, clients should respect a number of conventions in order to work properly. Clients must also be aware of possible effects of synchronization between input method and library in the case of a remote input server.

7.1. Client Conventions.

A well-behaved client (or toolkit) should first query the input method style. If the client cannot satisfy the requirements of the supported styles (in terms of geometry management or callbacks), it should negotiate with the user continuation of the program, or raise an exception or error of some sort.

Of course, clients using a toolkit including an on-the-spot text editing facility of some sort do not need to specify such callbacks.

7.2. Conventions for Input Method Synchronization

A KeyPress event with a keycode value of 0 shall be used exclusively as a signal that an input method has composed input which can be return by X*LookupString. No other use shall be made of a Keypress event with keycode 0.

A front-end input method shall signal to clients that they have composed input by sending a *synthetic KeyPress event* to the focus window of the corresponding XIC, with a keycode value of 0.

A back-end input method may change the keycode value of a KeyPress event to keycode 0 to indicate that there is data to be gathered from the filter.

When callback support is specified by client, input methods will not take action unless they explicitly called back the client and obtained no response (the callback is not specified, or returned invalid data).

8. Acknowledgments

This specification was generated after discussion on the mltalk mailing list, in several meetings, and several proposals were made by different entities or people. The contributors to the mltalk group have been:

Tim Anderson (Motorola), Alka Badshah (OSF), Gabe Beged-Dov (HP), Chih-Chung Ko (III), Vera Cheng (III), Michael Collins (Digital Equipment Co.), Walt Daniels (IBM), Noritoshi Demizu (OMRON), Keisuke Fukui (Fujitsu), Hitoshoi Fukumoto (Nihon - Sun Microsystems), Tim Greenwood (Digital Equipment Co.), John Harvey (IBM), Fred Horman (AT&T), Vania Joloboff (OSF), Norikazu Kaiya (Fujitsu), Yuji Kamata (IBM), Yutaka Kataoka (OMRON), Ranee Khubchandani (Sun Microsystems), Akari Kon (NEC), Hiroshi Kuribayashi (OMRON), Teruhiko Kurosaka (Sun Microsystems), Seiji Kuwari (Omron), Sandra Martin (OSF), Bill McMahon (HP), Tom McFarland (HP), Masato Morisaki (NTT), Nelson Ng (Sun), Takashi Nishimura (NTT America), Makato Nishino (IBM), Akira Ohsone (Nihon Sun), Chris Peterson (MIT), Sam Shteingart (AT&T), Manish Sheth (AT&T), Bob Scheifler (MIT), Muneyoshi Suzuki (NTT), Cori Mehring (Digital Equipment Co.), Frank Rojas (IBM), Shoji Sugiyama (IBM), Eiji Tosa (IBM) Glenn Widener (Tektronix).