Inter-Client Communication Conventions Manual

X Consortium Standard

David Rosenthal, Sun Microsystems, Inc.
Edited by Stuart W. Marks

Inter-Client Communication Conventions Manual: X Consortium Standard

by David Rosenthal and Stuart W. Marks

X Version 11, Release 7.7

Version 2.0

Copyright © 1988, 1991, 1993, 1994 X Consortium

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE X CONSORTIUM BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of the X Consortium shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from the X Consortium.

X Window System is a trademark of The Open Group.

Copyright © 1987, 1988, 1989, 1993, 1994 Sun Microsystems, Inc

Permission to use, copy, modify, and distribute this documentation for any purpose and without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. Sun Microsystems makes no representations about the suitability for any purpose of the information in this document. This documentation is provided as is without express or implied warranty.

Table of Contents

Preface to Version 2.0 v
Preface to Version 1.1 vi
1. Introduction 1
  - Evolution of the Conventions 1
  - Atoms 2
    - What Are Atoms? 2
    - Predefined Atoms 2
    - Naming Conventions 2
    - Semantics 3
    - Name Spaces 3
    - Discriminated Names 3

2. Peer-to-Peer Communication by Means of Selections 5
  - Acquiring Selection Ownership 5
  - Responsibilities of the Selection Owner 7
  - Giving Up Selection Ownership 9
    - Voluntarily Giving Up Selection Ownership 9
    - Forcibly Giving Up Selection Ownership 9
  - Requesting a Selection 10
  - Large Data Transfers 12
  - Use of Selection Atoms 13
    - Selection Atoms 13
    - Target Atoms 15
    - Selection Targets with Side Effects 17
  - Use of Selection Properties 18
    - TEXT Properties 19
    - INCR Properties 20
    - DRAWABLE Properties 21
    - SPAN Properties 21
  - Manager Selections 21

3. Peer-to-Peer Communication by Means of Cut Buffers 24
4. Client-to-Window-Manager Communication 25
  - Client's Actions 25
    - Creating a Top-Level Window 26
    - Client Properties 26
    - Window Manager Properties 34
    - Changing Window State 36
    - Configuring the Window 38
    - Changing Window Attributes 40
    - Input Focus 41
    - Colormaps 43
    - Icons 46
    - Pop-up Windows 47
    - Window Groups 48
  - Client Responses to Window Manager Actions 48
    - Reparenting 48
    - Redirection of Operations 49
    - Window Move 50
    - Window Resize 51
    - Iconify and Deiconify 51
    - Colormap Change 51
    - Input Focus 51

ClientMessage Events 52
Redirecting Requests 53
Communication with the Window Manager by Means of Selections 54
Summary of Window Manager Property Types 54
5. Session Management and Additional Inter-Client Exchanges 56
Client Support for Session Management 56
Window Manager Support for Session Management 57
Support for ICE Client Rendezvous 57
6. Manipulation of Shared Resources 58
The Input Focus 58
The Pointer 58
Grabs 58
Colormaps 60
The Keyboard Mapping 61
The Modifier Mapping 61
7. Device Color Characterization 64
XYZ <-> RGB Conversion Matrices 65
Intensity (dA RGB Value Conversion 65
8. Conclusion 68
The X Registry 68
A. Revision History 69
The X11R2 Draft 69
The July 27, 1988, Draft 69
The Public Review Drafts 69
Version 1.0, July 1989 70
Version 1.1 71
Public Review Draft, December 1993 71
Version 2.0, April 1994 73
B. Suggested Protocol Revisions 74
C. Obsolete Session Manager Conventions 75
Properties 75
WM_COMMAND Property 75
WM_CLIENT_MACHINE Property 75
Termination 76
Client Responses to Session Manager Actions 76
Saving Client State 76
Window Deletion 77
Summary of Session Manager Property Types 77

Preface to Version 2.0

The goal of the ICCCM Version 2.0 effort was to add new facilities, to fix problems with earlier drafts, and to improve readability and understandability, while maintaining compatibility with the earlier versions. This document is the product of over two years of discussion among the members of the X Consortium's `wmtalk` working group. The following people deserve thanks for their contributions:

Gabe Beged-Dov
Chan Benson
Jordan Brown
Larry Cable
Ellis Cohen
Donna Converse
Brian Cripe
Susan Dahlberg
Peter Daifuku
Andrew deBlois
Clive Feather
Stephen Gildea
Christian Jacobi
Bill Janssen
Vania Joloboff
Phil Karlton
Kaleb Keithley
Mark Manasse
Ralph Mor
Todd Newman
Bob Scheifler
Keith Taylor
Jim VanGilder
Mike Wexler
Michael Yee

It has been a privilege for me to work with this fine group of people.

Stuart W. Marks

December 1993

Preface to Version 1.1

David Rosenthal had overall architectural responsibility for the conventions defined in this document; he wrote most of the text and edited the document, but its development has been a communal effort. The details were thrashed out in meetings at the January 1988 MIT X Conference and at the 1988 Summer Usenix conference, and through months (and megabytes) of argument on the `wmtalk` mail alias. Thanks are due to everyone who contributed, and especially to the following people.

For the Selection section:

Jerry Farrell
Phil Karlton
Loretta Guarino Reid
Mark Manasse
Bob Scheifler

For the Cut-Buffer section:

Andrew Palay

For the Window and Session Manager sections:

Todd Brunhoff
Ellis Cohen
Jim Fulton
Hania Gajewska
Jordan Hubbard
Kerry Kimbrough
Audrey Ishizaki
Matt Landau
Mark Manasse
Bob Scheifler
Ralph Swick
Mike Wexler
Glenn Widener

For the Device Color Characterization section:

Keith Packard

In addition, thanks are due to those who contributed to the public review:

Gary Combs
Errol Crary
Nancy Cyprych
John Diamant
Clive Feather
Burns Fisher
Richard Greco
Tim Greenwood
Kee Hinckley
Brian Holt
John Interrante
John Irwin
Vania Joloboff
John Laporta
Ken Lee
Stuart Marks
Alan Mimms
Colas Nahaboo
Mark Patrick
Steve Pitschke
Brad Reed
John Thomas

Chapter 1. Introduction

It was an explicit design goal of X Version 11 to specify mechanism, not policy. As a result, a client that converses with the server using the protocol defined by the *X Window System Protocol, Version 11* may operate correctly in isolation but may not coexist properly with others sharing the same server.

Being a good citizen in the X Version 11 world involves adhering to conventions that govern inter-client communications in the following areas:

- Selection mechanism
- Cut buffers
- Window manager
- Session manager
- Manipulation of shared resources
- Device color characterization

This document proposes suitable conventions without attempting to enforce any particular user interface. To permit clients written in different languages to communicate, these conventions are expressed solely in terms of protocol operations, not in terms of their associated Xlib interfaces, which are probably more familiar. The binding of these operations to the Xlib interface for C and to the equivalent interfaces for other languages is the subject of other documents.

Evolution of the Conventions

In the interests of timely acceptance, the *Inter-Client Communication Conventions Manual* (ICCCM) covers only a minimal set of required conventions. These conventions will be added to and updated as appropriate, based on the experiences of the X Consortium.

As far as possible, these conventions are upwardly compatible with those in the February 25, 1988, draft that was distributed with the X Version 11, Release 2, of the software. In some areas, semantic problems were discovered with those conventions, and, thus, complete upward compatibility could not be assured. These areas are noted in the text and are summarized in Appendix A.

In the course of developing these conventions, a number of minor changes to the protocol were identified as desirable. They also are identified in the text, are summarized in Appendix B, and are offered as input to a future protocol revision process. If and when a protocol revision incorporating these changes is undertaken, it is anticipated that the ICCCM will need to be revised. Because it is difficult to ensure that clients and servers are upgraded simultaneously, clients using the revised conventions should examine the minor protocol revision number and be prepared to use the older conventions when communicating with an older server.

It is expected that these revisions will ensure that clients using the conventions appropriate to protocol minor revision *n* will interoperate correctly with those that use the conventions appropriate to protocol minor revision $n + 1$ if the server supports both.

Atoms

Many of the conventions use atoms. To assist the reader, the following sections attempt to amplify the description of atoms that is provided in the protocol specification.

What Are Atoms?

At the conceptual level, atoms are unique names that clients can use to communicate information to each other. They can be thought of as a bundle of octets, like a string but without an encoding being specified. The elements are not necessarily ASCII characters, and no case folding happens. [1]

The protocol designers felt that passing these sequences of bytes back and forth across the wire would be too costly. Further, they thought it important that events as they appear on the wire have a fixed size (in fact, 32 bytes) and that because some events contain atoms, a fixed-size representation for them was needed.

To allow a fixed-size representation, a protocol request (`InternAtom`) was provided to register a byte sequence with the server, which returns a 32-bit value (with the top three bits zero) that maps to the byte sequence. The inverse operator is also available (`GetAtomName`).

Predefined Atoms

The protocol specifies a number of atoms as being predefined:

> Predefined atoms are not strictly necessary and may not be useful in all environments, but they will eliminate many `InternAtom` requests in most applications. Note that they are predefined only in the sense of having numeric values, not in the sense of having required semantics.

Predefined atoms are an implementation trick to avoid the cost of interning many of the atoms that are expected to be used during the startup phase of all applications. The results of the `InternAtom` requests, which require a handshake, can be assumed *a priori*.

Language interfaces should probably cache the atom-name mappings and get them only when required. The CLX interface, for instance, makes no distinction between predefined atoms and other atoms; all atoms are viewed as symbols at the interface. However, a CLX implementation will typically keep a symbol or atom cache and will typically initialize this cache with the predefined atoms.

Naming Conventions

The built-in atoms are composed of uppercase ASCII characters with the logical words separated by an underscore character (_), for example, WM_ICON_NAME.

---

[1] The comment in the protocol specification for `InternAtom` that ISO Latin-1 encoding should be used is in the nature of a convention; the server treats the string as a byte sequence.

The protocol specification recommends that atoms used for private vendor-specific reasons should begin with an underscore. To prevent conflicts among organizations, additional prefixes should be chosen (for example, _DEC_WM_DECORATION_GEOMETRY).

The names were chosen in this fashion to make it easy to use them in a natural way within LISP. Keyword constructors allow the programmer to specify the atoms as LISP atoms. If the atoms were not all uppercase, special quoting conventions would have to be used.

Semantics

The core protocol imposes no semantics on atoms except as they are used in FONTPROP structures. For further information on FONTPROP semantics, see the *X Logical Font Description Conventions*.

Name Spaces

The protocol defines six distinct spaces in which atoms are interpreted. Any particular atom may or may not have some valid interpretation with respect to each of these name spaces.

Space	**Briefly**	**Examples**
Property name	Name	WM_HINTS, WM_NAME, RGB_BEST_MAP, ...
Property type	Type	WM_HINTS, CURSOR, RGB_COLOR_MAP, ...
Selection name	Selection	PRIMARY, SECONDARY, CLIPBOARD
Selection target	Target	FILE_NAME, POSTSCRIPT, PIXMAP, ...
Font property		QUAD_WIDTH, POINT_SIZE, ...
`ClientMessage` type		WM_SAVE_YOURSELF, _DEC_SAVE_EDITS, &...

Discriminated Names

Sometimes a protocol requires an arbitrary number of similar objects that need unique names (usually because the objects are created dynamically, so that names cannot be invented in advance). For example, a colormap-generating program might use the selection mechanism to offer colormaps for each screen and so needs a selection name for each screen. Such names are called "discriminated names" and are discriminated by some entity. This entity can be:

```
A screen
An X resource (a window, a colormap, a visual, etc.)
A client
```

If it is only necessary to generate a fixed set of names for each value of the discriminating entity, then the discriminated names are formed by suffixing an ordinary name according to the value of the entity.

If *name* is a descriptive portion for the name, *d* is a decimal number with no leading zeroes, and *x* is a hexadecimal number with exactly 8 digits, and using uppercase letters, then such discriminated names shall have the form:

Name Discriminated by	**Form**	**Example**
screen number	*name*_S*d*	WM_COMMS_S2
X resource	*name*_R*x*	GROUP_LEADER_R1234ABCD

To discriminate a name by client, use an X resource ID created by that client. This resource can be of any type.

Sometimes it is simply necessary to generate a unique set of names (for example, for the properties on a window used by a MULTIPLE selection). These names should have the form:

```
Ud     (e.g.,  U0  U1  U2  U3  ...)
```

if the names stand totally alone, and the form:

```
name_Ud     (e.g.,  FOO_U0  BAR_U0  FOO_U1  BAR_U1  ...)
```

if they come in sets (here there are two sets, named "FOO" and "BAR"). The standalone U*d* form should be used only if it is clear that the module using it has complete control over the relevant namespace or has the active cooperation of all other entities that might also use these names. (Naming properties on a window created specifically for a particular selection is such a use; naming properties on the root window is almost certainly not.)

In a particularly difficult case, it might be necessary to combine both forms of discrimination. If this happens, the U form should come after the other form, thus:

```
FOO_R12345678_U23
```

Rationale

Existing protocols will not be changed to use these naming conventions, because doing so will cause too much disruption. However, it is expected that future protocols -- both standard and private -- will use these conventions.

Chapter 2. Peer-to-Peer Communication by Means of Selections

Selections are the primary mechanism that X Version 11 defines for the exchange of information between clients, for example, by cutting and pasting between windows. Note that there can be an arbitrary number of selections (each named by an atom) and that they are global to the server. Use of Selection Atoms. discusses the choice of an atom. Each selection is owned by a client and is attached to a window.

Selections communicate between an owner and a requestor. The owner has the data representing the value of its selection, and the requestor receives it. A requestor wishing to obtain the value of a selection provides the following:

- The name of the selection
- The name of a property
- A window
- The atom representing the data type required
- Optionally, some parameters for the request

If the selection is currently owned, the owner receives an event and is expected to do the following:

- Convert the contents of the selection to the requested data type
- Place this data in the named property on the named window
- Send the requestor an event to let it know the property is available

Clients are strongly encouraged to use this mechanism. In particular, displaying text in a permanent window without providing the ability to select and convert it into a string is definitely considered antisocial.

Note that all data transferred between an owner and a requestor must usually go by means of the server in an X Version 11 environment. A client cannot assume that another client can open the same files or even communicate directly. The other client may be talking to the server by means of a completely different networking mechanism (for example, one client might be DECnet and the other TCP/IP). Thus, passing indirect references to data (such as, file names, host names, and port numbers) is permitted only if both clients specifically agree.

Acquiring Selection Ownership

A client wishing to acquire ownership of a particular selection should call `SetSelectionOwner`, which is defined as follows:

`SetSelectionOwner`

selection: ATOM

owner: WINDOW or `None`

time: TIMESTAMP or `CurrentTime`

The client should set the specified selection to the atom that represents the selection, set the specified owner to some window that the client created, and set the specified time to some time between the current last-change time of the selection concerned and the current server time. This time value usually will be obtained from the timestamp of the event that triggers the acquisition of the selection. Clients should not set the time value to `CurrentTime`, because if they do so, they have no way of finding when they gained ownership of the selection. Clients must use a window they created so that requestors can route events to the owner of the selection.[1]

Convention

Clients attempting to acquire a selection must set the time value of the `SetSelectionOwner` request to the timestamp of the event triggering the acquisition attempt, not to `CurrentTime`. A zero-length append to a property is a way to obtain a timestamp for this purpose; the timestamp is in the corresponding `PropertyNotify` event.

If the time in the `SetSelectionOwner` request is in the future relative to the server's current time or is in the past relative to the last time the specified selection changed hands, the `SetSelectionOwner` request appears to the client to succeed, but ownership is not actually transferred.

Because clients cannot name other clients directly, the specified owner window is used to refer to the owning client in the replies to `GetSelectionOwner`, in `SelectionRequest` and `SelectionClear` events, and possibly as a place to put properties describing the selection in question. To discover the owner of a particular selection, a client should invoke `GetSelectionOwner`, which is defined as follows:

`GetSelectionOwner`

selection: ATOM

->

owner: WINDOW or `None`

Convention

Clients are expected to provide some visible confirmation of selection ownership. To make this feedback reliable, a client must perform a sequence like the following:

```
SetSelectionOwner(selection=PRIMARY, owner=Window, time=timestamp)
owner = GetSelectionOwner(selection=PRIMARY)
if (owner != Window) Failure
```

---

[1] At present, no part of the protocol requires requestors to send events to the owner of a selection. This restriction is imposed to prepare for possible future extensions.

If the `SetSelectionOwner` request succeeds (not merely appears to succeed), the client that issues it is recorded by the server as being the owner of the selection for the time period starting at the specified time.

Responsibilities of the Selection Owner

When a requestor wants the value of a selection, the owner receives a `SelectionRequest` event, which is defined as follows:

`SelectionRequest`

owner: WINDOW
selection: ATOM
selection: ATOM
target: ATOM
property: ATOM or `None`
requestor: WINDOW
time: TIMESTAMP or `CurrentTime`

The specified owner and selection will be the values that were specified in the `SetSelectionOwner` request. The owner should compare the timestamp with the period it has owned the selection and, if the time is outside, refuse the `SelectionRequest` by sending the requestor window a `SelectionNotify` event with the property set to `None` (by means of a `SendEvent` request with an empty event mask).

More advanced selection owners are free to maintain a history of the value of the selection and to respond to requests for the value of the selection during periods they owned it even though they do not own it now.

If the specified property is `None`, the requestor is an obsolete client. Owners are encouraged to support these clients by using the specified target atom as the property name to be used for the reply.

Otherwise, the owner should use the target to decide the form into which the selection should be converted. Some targets may be defined such that requestors can pass parameters along with the request. The owner will find these parameters in the property named in the selection request. The type, format, and contents of this property are dependent upon the definition of the target. If the target is not defined to have parameters, the owner should ignore the property if it is present. If the selection cannot be converted into a form based on the target (and parameters, if any), the owner should refuse the `SelectionRequest` as previously described.

If the specified property is not `None`, the owner should place the data resulting from converting the selection into the specified property on the requestor window and should set the property's type to some appropriate value, which need not be the same as the specified target.

Convention

All properties used to reply to `SelectionRequest` events must be placed on the requestor window.

In either case, if the data comprising the selection cannot be stored on the requestor window (for example, because the server cannot provide sufficient memory), the owner must refuse the `SelectionRequest`, as previously described. See also Large Data Transfers.

If the property is successfully stored, the owner should acknowledge the successful conversion by sending the requestor window a `SelectionNotify` event (by means of a `SendEvent` request with an empty mask). `SelectionNotify` is defined as follows:

`SelectionNotify`

requestor: WINDOW

selection, *target*: ATOM

property: ATOM or `None`

time: TIMESTAMP or `CurrentTime`

The owner should set the specified selection, target, time, and property arguments to the values received in the `SelectionRequest` event. (Note that setting the property argument to `None` indicates that the conversion requested could not be made.)

Convention

The selection, target, time, and property arguments in the `SelectionNotify` event should be set to the values received in the `SelectionRequest` event.

If the owner receives more than one `SelectionRequest` event with the same requestor, selection, target, and timestamp it must respond to them in the same order in which they were received.

Rationale

It is possible for a requestor to have multiple outstanding requests that use the same requestor window, selection, target, and timestamp, and that differ only in the property. If this occurs, and one of the conversion requests fails, the resulting `SelectionNotify` event will have its property argument set to `None`. This may make it impossible for the requestor to determine which conversion request had failed, unless the requests are responded to in order.

The data stored in the property must eventually be deleted. A convention is needed to assign the responsibility for doing so.

Convention

Selection requestors are responsible for deleting properties whose names they receive in `SelectionNotify` events (See Requesting a Selection) or in properties with type MULTIPLE.

A selection owner will often need confirmation that the data comprising the selection has actually been transferred. (For example, if the operation has side effects on the owner's internal data structures, these should not take place until the requestor has indicated that it has successfully received the data.) Owners should express interest in `PropertyNotify` events for the specified requestor window and wait until the property in the `SelectionNotify` event has been deleted before assuming that the selection data has been transferred. For the MULTIPLE request, if the different conversions require separate confirmation, the selection owner can also watch for the deletion of the individual properties named in the property in the `SelectionNotify` event.

When some other client acquires a selection, the previous owner receives a `SelectionClear` event, which is defined as follows:

`SelectionClear`

owner: WINDOW
selection: ATOM
time: TIMESTAMP

The timestamp argument is the time at which the ownership changed hands, and the owner argument is the window the previous owner specified in its `SetSelectionOwner` request.

If an owner loses ownership while it has a transfer in progress (that is, before it receives notification that the requestor has received all the data), it must continue to service the ongoing transfer until it is complete.

If the selection value completely changes, but the owner happens to be the same client (for example, selecting a totally different piece of text in the same `xterm` as before), then the client should reacquire the selection ownership as if it were not the owner, providing a new timestamp. If the selection value is modified, but can still reasonably be viewed as the same selected object, [2] the owner should take no action.

Giving Up Selection Ownership

Clients may either give up selection ownership voluntarily or lose it forcibly as the result of some other client's actions.

Voluntarily Giving Up Selection Ownership

To relinquish ownership of a selection voluntarily, a client should execute a `SetSelectionOwner` request for that selection atom, with owner specified as `None` and the time specified as the timestamp that was used to acquire the selection.

Alternatively, the client may destroy the window used as the owner value of the `SetSelectionOwner` request, or the client may terminate. In both cases, the ownership of the selection involved will revert to `None`.

Forcibly Giving Up Selection Ownership

If a client gives up ownership of a selection or if some other client executes a `SetSelectionOwner` for it and thus reassigns it forcibly, the previous owner will receive a

---

[2] The division between these two cases is a matter of judgment on the part of the software developer.

`SelectionClear` event. For the definition of a `SelectionClear` event, see Responsibilities of the Selection Owner

The timestamp is the time the selection changed hands. The specified owner is the window that was specified by the current owner in its `SetSelectionOwner` request.

Requesting a Selection

A client that wishes to obtain the value of a selection in a particular form (the requestor) issues a `ConvertSelection` request, which is defined as follows:

`ConvertSelection`

selection, *target*: ATOM

property: ATOM or `None`

requestor: WINDOW

time: TIMESTAMP or `CurrentTime`

The selection argument specifies the particular selection involved, and the target argument specifies the required form of the information. For information about the choice of suitable atoms to use, see Use of Selection Atoms The requestor should set the requestor argument to a window that it created; the owner will place the reply property there. The requestor should set the time argument to the timestamp on the event that triggered the request for the selection value. Note that clients should not specify `CurrentTime`.

Convention

Clients should not use `CurrentTime` for the time argument of a `ConvertSelection` request. Instead, they should use the timestamp of the event that caused the request to be made.

The requestor should set the property argument to the name of a property that the owner can use to report the value of the selection. Requestors should ensure that the named property does not exist on the window before issuing the `ConvertSelection` request.[3] The exception to this rule is when the requestor intends to pass parameters with the request (see below).

Rationale

It is necessary for requestors to delete the property before issuing the request so that the target can later be extended to take parameters without introducing an incompatibility. Also note that the requestor of a selection need not know the client that owns the selection nor the window on which the selection was acquired.

Some targets may be defined such that requestors can pass parameters along with the request. If the requestor wishes to provide parameters to a request, they should be placed in the specified property on the requestor window before the requestor

---

[3] This requirement is new in version 2.0, and, in general, existing clients do not conform to this requirement. To prevent these clients from breaking, no existing targets should be extended to take parameters until sufficient time has passed for clients to be updated. Note that the MULTIPLE target was defined to take parameters in version 1.0 and its definition is not changing. There is thus no conformance problem with MULTIPLE.

issues the `ConvertSelection` request, and this property should be named in the request.

Some targets may be defined so that parameters are optional. If no parameters are to be supplied with the request of such a target, the requestor must ensure that the property does not exist before issuing the `ConvertSelection` request.

The protocol allows the property field to be set to `None`, in which case the owner is supposed to choose a property name. However, it is difficult for the owner to make this choice safely.

Conventions

- Requestors should not use `None` for the property argument of a `ConvertSelection` request.
- Owners receiving `ConvertSelection` requests with a property argument of `None` are talking to an obsolete client. They should choose the target atom as the property name to be used for the reply.

The result of the `ConvertSelection` request is that a `SelectionNotify` event will be received. For the definition of a `SelectionNotify` event, see Responsibilities of the Selection Owner.

The requestor, selection, time, and target arguments will be the same as those on the `ConvertSelection` request.

If the property argument is `None`, the conversion has been refused. This can mean either that there is no owner for the selection, that the owner does not support the conversion implied by the target, or that the server did not have sufficient space to accommodate the data.

If the property argument is not `None`, then that property will exist on the requestor window. The value of the selection can be retrieved from this property by using the `GetProperty` request, which is defined as follows:

`GetProperty`

window: WINDOW

property: ATOM

type: ATOM or `AnyPropertyType`

long-offset, *long-length*: CARD32

delete: BOOL

->

type: ATOM or `None`

format: {0, 8, 16, 32}

bytes-after: CARD32

value: LISTofINT8 or LISTofINT16 or LISTofINT32

`GetProperty` to retrieve the value of a selection, the property argument should be set to the corresponding value in the `SelectionNotify` event. Because the requestor has no way of knowing beforehand what type the selection owner will use, the type argument should be set to `AnyPropertyType`. Several `GetProperty` requests may be needed to retrieve all the data in the selection; each should set the long-offset argument to the amount of data received so far, and the size argument to some reasonable buffer size (see Large Data Transfers.). If the returned value of bytes-after is zero, the whole property has been transferred.

Once all the data in the selection has been retrieved (which may require getting the values of several properties -- see Use of Selection Properties.), the requestor should delete the property in the `SelectionNotify` request by using a `GetProperty` request with the delete argument set to `True`. As previously discussed, the owner has no way of knowing when the data has been transferred to the requestor unless the property is removed.

Convention

The requestor must delete the property named in the `SelectionNotify` once all the data has been retrieved. The requestor should invoke either `DeleteProperty` or `GetProperty` (delete==True) after it has successfully retrieved all the data in the selection. For further information, see Large Data Transfers.

Large Data Transfers

Selections can get large, which poses two problems:

- Transferring large amounts of data to the server is expensive.
- All servers will have limits on the amount of data that can be stored in properties. Exceeding this limit will result in an `Alloc` error on the `ChangeProperty` request that the selection owner uses to store the data.

The problem of limited server resources is addressed by the following conventions:

Conventions

- Selection owners should transfer the data describing a large selection (relative to the maximum-request-size they received in the connection handshake) using the INCR property mechanism (see INCR Properties.).
- Any client using `SetSelectionOwner` to acquire selection ownership should arrange to process `Alloc` errors in property change requests. For clients using Xlib, this involves using the `XSetErrorHandler` function to override the default handler.
- A selection owner must confirm that no `Alloc` error occurred while storing the properties for a selection before replying with a confirming `SelectionNotify` event.
- When storing large amounts of data (relative to maximum-request-size), clients should use a sequence of `ChangeProperty (mode==Append)` requests for reasonable quantities of data. This avoids locking servers up and limits the waste of data an `Alloc` error would cause.
- If an `Alloc` error occurs during the storing of the selection data, all properties stored for this selection should be deleted and the `ConvertSelection` request should be refused (see Responsibilities of the Selection Owner.).

- To avoid locking servers up for inordinate lengths of time, requestors retrieving large quantities of data from a property should perform a series of `GetProperty` requests, each asking for a reasonable amount of data.

Advice to Implementors

Single-threaded servers should take care to avoid locking up during large data transfers.

Use of Selection Atoms

Defining a new atom consumes resources in the server that are not released until the server reinitializes. Thus, reducing the need for newly minted atoms is an important goal for the use of the selection atoms.

Selection Atoms

There can be an arbitrary number of selections, each named by an atom. To conform with the inter-client conventions, however, clients need deal with only these three selections:

- PRIMARY
- SECONDARY
- CLIPBOARD

Other selections may be used freely for private communication among related groups of clients.

The PRIMARY Selection

The selection named by the atom PRIMARY is used for all commands that take only a single argument and is the principal means of communication between clients that use the selection mechanism.

The SECONDARY Selection

The selection named by the atom SECONDARY is used:

- As the second argument to commands taking two arguments (for example, "exchange primary and secondary selections")
- As a means of obtaining data when there is a primary selection and the user does not want to disturb it

The CLIPBOARD Selection

The selection named by the atom CLIPBOARD is used to hold data that is being transferred between clients, that is, data that usually is being cut and then pasted or copied and then pasted. Whenever a client wants to transfer data to the clipboard:

- It should assert ownership of the CLIPBOARD.

- If it succeeds in acquiring ownership, it should be prepared to respond to a request for the contents of the CLIPBOARD in the usual way (retaining the data to be able to return it). The request may be generated by the clipboard client described below.

- If it fails to acquire ownership, a cutting client should not actually perform the cut or provide feedback that would suggest that it has actually transferred data to the clipboard.

The owner should repeat this process whenever the data to be transferred would change.

Clients wanting to paste data from the clipboard should request the contents of the CLIPBOARD selection in the usual way.

Except while a client is actually deleting or copying data, the owner of the CLIPBOARD selection may be a single, special client implemented for the purpose. This client maintains the content of the clipboard up-to-date and responds to requests for data from the clipboard as follows:

- It should assert ownership of the CLIPBOARD selection and reassert it any time the clipboard data changes.

- If it loses the selection (because another client has some new data for the clipboard), it should:

  - Obtain the contents of the selection from the new owner by using the timestamp in the `SelectionClear` event.

  - Attempt to reassert ownership of the CLIPBOARD selection by using the same timestamp.

  - Restart the process using a newly acquired timestamp if this attempt fails. This timestamp should be obtained by asking the current owner of the CLIPBOARD selection to convert it to a TIMESTAMP. If this conversion is refused or if the same timestamp is received twice, the clipboard client should acquire a fresh timestamp in the usual way (for example by a zero-length append to a property).

- It should respond to requests for the CLIPBOARD contents in the usual way.

A special CLIPBOARD client is not necessary. The protocol used by the cutting client and the pasting client is the same whether the CLIPBOARD client is running or not. The reasons for running the special client include:

- Stability - If the cutting client were to crash or terminate, the clipboard value would still be available.

- Feedback - The clipboard client can display the contents of the clipboard.

- Simplicity - A client deleting data does not have to retain it for so long, thus reducing the chance of race conditions causing problems.

The reasons not to run the clipboard client include:

- Performance - Data is transferred only if it is actually required (that is, when some client actually wants the data).

- Flexibility - The clipboard data may be available as more than one target.

Target Atoms

The atom that a requestor supplies as the target of a `ConvertSelection` request determines the form of the data supplied. The set of such atoms is extensible, but a generally accepted base set of target atoms is needed. As a starting point for this, the following table contains those that have been suggested so far.

Atom	**Type**	**Data Received**
ADOBE_PORTABLE_DOCUMENT_FORMAT	STRING	[1]
APPLE_PICT	APPLE_PICT	[2]
BACKGROUND	PIXEL	A list of pixel values
BITMAP	BITMAP	A list of bitmap IDs
CHARACTER_POSITION	SPAN	The start and end of the selection in bytes
CLASS	TEXT	(see WM_CLASS Property.)
CLIENT_WINDOW	WINDOW	Any top-level window owned by the selection owner
COLORMAP	COLORMAP	A list of colormap IDs
COLUMN_NUMBER	SPAN	The start and end column numbers
COMPOUND_TEXT	COMPOUND_TEXT	Compound Text
DELETE	NULL	(see DELETE.)
DRAWABLE	DRAWABLE	A list of drawable IDs
ENCAPSULATED_POSTSCRIPT	STRING	[3], Appendix H [a]
ENCAPSULATED_POSTSCRIPT_INTERCHANGE	STRING	[3], Appendix H
FILE_NAME	TEXT	The full path name of a file
FOREGROUND	PIXEL	A list of pixel values
HOST_NAME	TEXT	(see WM_CLIENT_MACHINE Property.)
INSERT_PROPERTY	NULL	(see INSERT_PROPERTY.)
INSERT_SELECTION	NULL	(see INSERT_SELECTION.)
LENGTH	INTEGER	The number of bytes in the selection [b]
LINE_NUMBER	SPAN	The start and end line numbers
LIST_LENGTH	INTEGER	The number of disjoint parts of the selection
MODULE	TEXT	The name of the selected procedure
MULTIPLE	ATOM_PAIR	(see the discussion that follows)

Atom	**Type**	**Data Received**
NAME	TEXT	(see WM_NAME Property.)
ODIF	TEXT	ISO Office Document Interchange Format
OWNER_OS	TEXT	The operating system of the owner client
PIXMAP	PIXMAP [c]	A list of pixmap IDs
POSTSCRIPT	STRING	[3]
PROCEDURE	TEXT	The name of the selected procedure
PROCESS	INTEGER, TEXT	The process ID of the owner
STRING	STRING	ISO Latin-1 (+TAB +NEWLINE) text
TARGETS	ATOM	A list of valid target atoms
TASK	INTEGER, TEXT	The task ID of the owner
TEXT	TEXT	The text in the owner's choice of encoding
TIMESTAMP	INTEGER	The timestamp used to acquire the selection
USER	TEXT	The name of the user running the owner

[a] Earlier versions of this document erroneously specified that conversion of the PIXMAP target returns a property of type DRAWABLE instead of PIXMAP. Implementors should be aware of this and may want to support the DRAWABLE type as well to allow for compatibility with older clients.

[b] The targets ENCAPSULATED_POSTSCRIPT and ENCAPSULATED_POSTSCRIPT_INTERCHANGE are equivalent to the targets _ADOBE_EPS and _ADOBE_EPSI (respectively) that appear in the selection targets registry. The _ADOBE_ targets are deprecated, but clients are encouraged to continue to support them for backward compatibility.

[c] This definition is ambiguous, as the selection may be converted into any of several targets that may return differing amounts of data. The requestor has no way of knowing which, if any, of these targets corresponds to the result of LENGTH. Clients are advised that no guarantees can be made about the result of a conversion to LENGTH; its use is thus deprecated.

References:

1. Adobe Systems, Incorporated. *Portable Document Format Reference Manual.* Reading, MA, Addison-Wesley, ISBN 0-201-62628-4.
2. Apple Computer, Incorporated. *Inside Macintosh, Volume V.* Chapter 4, "Color QuickDraw," Color Picture Format. ISBN 0-201-17719-6.
3. Adobe Systems, Incorporated. *PostScript Language Reference Manual.* Reading, MA, Addison-Wesley, ISBN 0-201-18127-4.

It is expected that this table will grow over time.

Selection owners are required to support the following targets. All other targets are optional.

- TARGETS - The owner should return a list of atoms that represent the targets for which an attempt to convert the current selection will succeed (barring unforseeable problems such as `Alloc` errors). This list should include all the required atoms.

- MULTIPLE - The MULTIPLE target atom is valid only when a property is specified on the `ConvertSelection` request. If the property argument in the `SelectionRequest` event is `None` and the target is MULTIPLE, it should be refused.

  When a selection owner receives a `SelectionRequest (target==MULTIPLE)` request, the contents of the property named in the request will be a list of atom pairs: the first atom naming a target and the second naming a property `( None` is not valid here). The effect should be as if the owner had received a sequence of `SelectionRequest` events (one for each atom pair) except that:

  - The owner should reply with a `SelectionNotify` only when all the requested conversions have been performed.
  - If the owner fails to convert the target named by an atom in the MULTIPLE property, it should replace that atom in the property with `None`.

  ### Convention

  The entries in a MULTIPLE property must be processed in the order they appear in the property. For further information, see Selection Targets with Side Effects.

  The requestor should delete each individual property when it has copied the data from that conversion, and the property specified in the MULTIPLE request when it has copied all the data.

  The requests are otherwise to be processed independently, and they should succeed or fail independently. The MULTIPLE target is an optimization that reduces the amount of protocol traffic between the owner and the requestor; it is not a transaction mechanism. For example, a client may issue a MULTIPLE request with two targets: a data target and the DELETE target. The DELETE target will still be processed even if the conversion of the data target fails.

- TIMESTAMP - To avoid some race conditions, it is important that requestors be able to discover the timestamp the owner used to acquire ownership. Until and unless the protocol is changed so that a `GetSelectionOwner` request returns the timestamp used to acquire ownership, selection owners must support conversion to TIMESTAMP, returning the timestamp they used to obtain the selection.

Selection Targets with Side Effects

Some targets (for example, DELETE) have side effects. To render these targets unambiguous, the entries in a MULTIPLE property must be processed in the order that they appear in the property.

In general, targets with side effects will return no information, that is, they will return a zero length property of type NULL. (Type NULL means the result of `InternAtom` on the string "NULL", not the value zero.) In all cases, the requested side effect must be performed before the conversion is accepted. If the requested side effect cannot be performed, the corresponding conversion request must be refused.

Conventions

- Targets with side effects should return no information (that is, they should have a zero-length property of type NULL).

- The side effect of a target must be performed before the conversion is accepted.
- If the side effect of a target cannot be performed, the corresponding conversion request must be refused.

Problem

The need to delay responding to the `ConvertSelection` request until a further conversion has succeeded poses problems for the Intrinsics interface that need to be addressed.

These side-effect targets are used to implement operations such as "exchange PRIMARY and SECONDARY selections."

DELETE

When the owner of a selection receives a request to convert it to DELETE, it should delete the corresponding selection (whatever doing so means for its internal data structures) and return a zero-length property of type NULL if the deletion was successful.

INSERT_SELECTION

When the owner of a selection receives a request to convert it to INSERT_SELECTION, the property named will be of type ATOM_PAIR. The first atom will name a selection, and the second will name a target. The owner should use the selection mechanism to convert the named selection into the named target and should insert it at the location of the selection for which it got the INSERT_SELECTION request (whatever doing so means for its internal data structures).

INSERT_PROPERTY

When the owner of a selection receives a request to convert it to INSERT_PROPERTY, it should insert the property named in the request at the location of the selection for which it got the INSERT_SELECTION request (whatever doing so means for its internal data structures).

Use of Selection Properties

The names of the properties used in selection data transfer are chosen by the requestor. The use of `None` property fields in `ConvertSelection` requests (which request the selection owner to choose a name) is not permitted by these conventions.

The selection owner always chooses the type of the property in the selection data transfer. Some types have special semantics assigned by convention, and these are reviewed in the following sections.

In all cases, a request for conversion to a target should return either a property of one of the types listed in the previous table for that target or a property of type INCR and then a property of one of the listed types.

Certain selection properties may contain resource IDs. The selection owner should ensure that the resource is not destroyed and that its contents are not changed until after the selection transfer is complete. Requestors that rely on the existence or on the proper contents of a resource must operate on the resource (for example, by copying the contents of a pixmap) before deleting the selection property.

The selection owner will return a list of zero or more items of the type indicated by the property type. In general, the number of items in the list will correspond to the number of disjoint parts of the selection. Some targets (for example, side-effect targets) will be of length zero irrespective of the number of disjoint selection parts. In the case of fixed-size items, the requestor may determine the number of items by the property size. Selection property types are listed in the table below. For variable-length items such as text, the separators are also listed.

Type Atom	**Format**	**Separator**
APPLE_PICT	8	Self-sizing
ATOM	32	Fixed-size
ATOM_PAIR	32	Fixed-size
BITMAP	32	Fixed-size
C_STRING	8	Zero
COLORMAP	32	Fixed-size
COMPOUND_TEXT	8	Zero
DRAWABLE	32	Fixed-size
INCR	32	Fixed-size
INTEGER	32	Fixed-size
PIXEL	32	Fixed-size
PIXMAP	32	Fixed-size
SPAN	32	Fixed-size
STRING	8	Zero
WINDOW	32	Fixed-size

It is expected that this table will grow over time.

TEXT Properties

In general, the encoding for the characters in a text string property is specified by its type. It is highly desirable for there to be a simple, invertible mapping between string property types and any character set names embedded within font names in any font naming standard adopted by the Consortium.

The atom TEXT is a polymorphic target. Requesting conversion into TEXT will convert into whatever encoding is convenient for the owner. The encoding chosen will be indicated by the type of the property returned. TEXT is not defined as a type; it will never be the returned type from a selection conversion request.

If the requestor wants the owner to return the contents of the selection in a specific encoding, it should request conversion into the name of that encoding.

In the table in Target Atoms, the word TEXT (in the Type column) is used to indicate one of the registered encoding names. The type would not actually be TEXT; it would be STRING or some other ATOM naming the encoding chosen by the owner.

STRING as a type or a target specifies the ISO Latin-1 character set plus the control characters TAB (octal 11) and NEWLINE (octal 12). The spacing interpretation of TAB is context dependent. Other ASCII control characters are explicitly not included in STRING at the present time.

COMPOUND_TEXT as a type or a target specifies the Compound Text interchange format; see the *Compound Text Encoding*.

There are some text objects where the source or intended user, as the case may be, does not have a specific character set for the text, but instead merely requires a zero-terminated sequence of bytes with no other restriction; no element of the selection mechanism may assume that any byte value is forbidden or that any two differing sequences are equivalent. [4] For these objects, the type C_STRING should be used.

Rationale

An example of the need for C_STRING is to transmit the names of files; many operating systems do not interpret filenames as having a character set. For example, the same character string uses a different sequence of bytes in ASCII and EBCDIC, and so most operating systems see these as different filenames and offer no way to treat them as the same. Thus no character-set based property type is suitable.

Type STRING, COMPOUND_TEXT, and C_STRING properties will consist of a list of elements separated by null characters; other encodings will need to specify an appropriate list format.

INCR Properties

Requestors may receive a property of type INCR [5] in response to any target that results in selection data.

This indicates that the owner will send the actual data incrementally. The contents of the INCR property will be an integer, which represents a lower bound on the number of bytes of data in the selection. The requestor and the selection owner transfer the data in the selection in the following manner.

The selection requestor starts the transfer process by deleting the (type==INCR) property forming the reply to the selection.

The selection owner then:

- Appends the data in suitable-size chunks to the same property on the same window as the selection reply with a type corresponding to the actual type of the converted selection. The size should be less than the maximum-request-size in the connection handshake.
- Waits between each append for a `PropertyNotify` (state==Deleted) event that shows that the requestor has read the data. The reason for doing this is to limit the consumption of space in the server.

---

[4] Note that this is different from STRING, where many byte values are forbidden, and from COMPOUND_TEXT, where, for example, inserting the sequence 27,\ 40,\ 66 (designate ASCII into GL) at the start does not alter the meaning.

[5] These properties were called INCREMENTAL in an earlier draft. The protocol for using them has changed, and so the name has changed to avoid confusion.

- Waits (after the entire data has been transferred to the server) until a `PropertyNotify` (state==Deleted) event that shows that the data has been read by the requestor and then writes zero-length data to the property.

The selection requestor:

- Waits for the `SelectionNotify` event.
- Loops:
  - Retrieving data using `GetProperty` with the delete argument `True`.
  - Waiting for a `PropertyNotify` with the state argument `NewValue`.
- Waits until the property named by the `PropertyNotify` event is zero-length.
- Deletes the zero-length property.

The type of the converted selection is the type of the first partial property. The remaining partial properties must have the same type.

DRAWABLE Properties

Requestors may receive properties of type PIXMAP, BITMAP, DRAWABLE, or WINDOW, which contain an appropriate ID. While information about these drawables is available from the server by means of the `GetGeometry` request, the following items are not:

- Foreground pixel
- Background pixel
- Colormap ID

In general, requestors converting into targets whose returned type in the table in Target Atoms is one of the DRAWABLE types should expect to convert also into the following targets (using the MULTIPLE mechanism):

- FOREGROUND returns a PIXEL value.
- BACKGROUND returns a PIXEL value.
- COLORMAP returns a colormap ID.

SPAN Properties

Properties with type SPAN contain a list of cardinal-pairs with the length of the cardinals determined by the format. The first specifies the starting position, and the second specifies the ending position plus one. The base is zero. If they are the same, the span is zero-length and is before the specified position. The units are implied by the target atom, such as LINE_NUMBER or CHARACTER_POSITION.

Manager Selections

Certain clients, often called managers, take on responsibility for managing shared resources. A client that manages a shared resource should take ownership of an appropriate selection, named using the conventions described in Naming Conventions and Discriminated Names. A client that manages multiple shared resources (or groups of resources) should take ownership of a selection for each one.

The manager may support conversion of various targets for that selection. Managers are encouraged to use this technique as the primary means by which clients interact with the managed resource. Note that the conventions for interacting with the window manager predate this section; as a result many interactions with the window manager use other techniques.

Before a manager takes ownership of a manager selection, it should use the `GetSelectionOwner` request to check whether the selection is already owned by another client, and, where appropriate, it should ask the user if the new manager should replace the old one. If so, it may then take ownership of the selection. Managers should acquire the selection using a window created expressly for this purpose. Managers must conform to the rules for selection owners described in Acquiring Selection Ownership and Responsibilities of the Selection Owner , and they must also support the required targets listed in Use of Selection Atoms.

If a manager loses ownership of a manager selection, this means that a new manager is taking over its responsibilities. The old manager must release all resources it has managed and must then destroy the window that owned the selection. For example, a window manager losing ownership of WM_S2 must deselect from `SubstructureRedirect` on the root window of screen 2 before destroying the window that owned WM_S2.

When the new manager notices that the window owning the selection has been destroyed, it knows that it can successfully proceed to control the resource it is planning to manage. If the old manager does not destroy the window within a reasonable time, the new manager should check with the user before destroying the window itself or killing the old manager.

If a manager wants to give up, on its own, management of a shared resource controlled by a selection, it must do so by releasing the resources it is managing and then by destroying the window that owns the selection. It should not first disown the selection, since this introduces a race condition.

Clients who are interested in knowing when the owner of a manager selection is no longer managing the corresponding shared resource should select for `StructureNotify` on the window owning the selection so they can be notified when the window is destroyed. Clients are warned that after doing a `GetSelectionOwner` and selecting for `StructureNotify`, they should do a `GetSelectionOwner` again to ensure that the owner did not change after initially getting the selection owner and before selecting for `StructureNotify`.

Immediately after a manager successfully acquires ownership of a manager selection, it should announce its arrival by sending a `ClientMessage` event. This event should be sent using the `SendEvent` protocol request with the following arguments:

Argument	**Value**
destination:	the root window of screen 0, or the root window of the appropriate screen if the manager is managing a screen-specific resource
propogate:	False

Argument	Value
event-mask:	`StructureNotify`
event:	`ClientMessage`
type:	MANAGER
format:	32
data[0] [a]	timestamp
data[1]:	manager selection atom
data[2]:	the window owning the selection
data[3]:	manager-selection-specific data
data[4]:	manager-selection-specific data

---

[a] We use the notation data[n] to indicate the n th element of the LISTofINT8, LISTofINT16, or LISTofINT32 in the data field of the `ClientMessage`, according to the format field. The list is indexed from zero.

Clients that wish to know when a specific manager has started should select for `StructureNotify` on the appropriate root window and should watch for the appropriate MANAGER `ClientMessage`.

Chapter 3. Peer-to-Peer Communication by Means of Cut Buffers

The cut buffer mechanism is much simpler but much less powerful than the selection mechanism. The selection mechanism is active in that it provides a link between the owner and requestor clients. The cut buffer mechanism is passive; an owner places data in a cut buffer from which a requestor retrieves the data at some later time.

The cut buffers consist of eight properties on the root of screen zero, named by the predefined atoms CUT_BUFFER0 to CUT_BUFFER7. These properties must, at present, have type STRING and format 8. A client that uses the cut buffer mechanism must initially ensure that all eight properties exist by using `ChangeProperty` requests to append zero-length data to each.

A client that stores data in the cut buffers (an owner) first must rotate the ring of buffers by plus 1 by using `RotateProperties` requests to rename each buffer; that is, CUT_BUFFER0 to CUT_BUFFER1, CUT_BUFFER1 to CUT_BUFFER2, ..., and CUT_BUFFER7 to CUT_BUFFER0. It then must store the data into CUT_BUFFER0 by using a `ChangeProperty` request in mode `Replace`.

A client that obtains data from the cut buffers should use a `GetProperty` request to retrieve the contents of CUT_BUFFER0.

In response to a specific user request, a client may rotate the cut buffers by minus 1 by using `RotateProperties` requests to rename each buffer; that is, CUT_BUFFER7 to CUT_BUFFER6, CUT_BUFFER6 to CUT_BUFFER5, ..., and CUT_BUFFER0 to CUT_BUFFER7.

Data should be stored to the cut buffers and the ring rotated only when requested by explicit user action. Users depend on their mental model of cut buffer operation and need to be able to identify operations that transfer data to and fro.

Chapter 4. Client-to-Window-Manager Communication

To permit window managers to perform their role of mediating the competing demands for resources such as screen space, the clients being managed must adhere to certain conventions and must expect the window managers to do likewise. These conventions are covered here from the client's point of view.

In general, these conventions are somewhat complex and will undoubtedly change as new window management paradigms are developed. Thus, there is a strong bias toward defining only those conventions that are essential and that apply generally to all window management paradigms. Clients designed to run with a particular window manager can easily define private protocols to add to these conventions, but they must be aware that their users may decide to run some other window manager no matter how much the designers of the private protocol are convinced that they have seen the "one true light" of user interfaces.

It is a principle of these conventions that a general client should neither know nor care which window manager is running or, indeed, if one is running at all. The conventions do not support all client functions without a window manager running; for example, the concept of Iconic is not directly supported by clients. If no window manager is running, the concept of Iconic does not apply. A goal of the conventions is to make it possible to kill and restart window managers without loss of functionality.

Each window manager will implement a particular window management policy; the choice of an appropriate window management policy for the user's circumstances is not one for an individual client to make but will be made by the user or the user's system administrator. This does not exclude the possibility of writing clients that use a private protocol to restrict themselves to operating only under a specific window manager. Rather, it merely ensures that no claim of general utility is made for such programs.

For example, the claim is often made: "The client I'm writing is important, and it needs to be on top." Perhaps it is important when it is being run in earnest, and it should then be run under the control of a window manager that recognizes "important" windows through some private protocol and ensures that they are on top. However, imagine, for example, that the "important" client is being debugged. Then, ensuring that it is always on top is no longer the appropriate window management policy, and it should be run under a window manager that allows other windows (for example, the debugger) to appear on top.

Client's Actions

In general, the object of the X Version 11 design is that clients should, as far as possible, do exactly what they would do in the absence of a window manager, except for the following:

- Hinting to the window manager about the resources they would like to obtain
- Cooperating with the window manager by accepting the resources they are allocated even if they are not those requested

- Being prepared for resource allocations to change at any time

Creating a Top-Level Window

A client's *top-level window* is a window whose override-redirect attribute is `False`. It must either be a child of a root window, or it must have been a child of a root window immediately prior to having been reparented by the window manager. If the client reparents the window away from the root, the window is no longer a top-level window; but it can become a top-level window again if the client reparents it back to the root.

A client usually would expect to create its top-level windows as children of one or more of the root windows by using some boilerplate like the following:

```
win = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), xsh.x, xsh.y,
     xsh.width, xsh.height, bw, bd, bg);
```

If a particular one of the root windows was required, however, it could use something like the following:

```
win = XCreateSimpleWindow(dpy, RootWindow(dpy, screen), xsh.x, xsh.y,
     xsh.width, xsh.height, bw, bd, bg);
```

Ideally, it should be possible to override the choice of a root window and allow clients (including window managers) to treat a nonroot window as a pseudo-root. This would allow, for example, the testing of window managers and the use of application-specific window managers to control the subwindows owned by the members of a related suite of clients. Doing so properly requires an extension, the design of which is under study.

From the client's point of view, the window manager will regard its top-level window as being in one of three states:

- Normal
- Iconic
- Withdrawn

Newly created windows start in the Withdrawn state. Transitions between states happen when the top-level window is mapped and unmapped and when the window manager receives certain messages. For further details, see WM_HINTS Property. and Changing Window State.

Client Properties

Once the client has one or more top-level windows, it should place properties on those windows to inform the window manager of the behavior that the client desires. Window managers will assume values they find convenient for any of these properties that are not supplied; clients that depend on particular values must explicitly supply them. The window manager will not change properties written by the client.

The window manager will examine the contents of these properties when the window makes the transition from the Withdrawn state and will monitor some properties for changes while the window is in the Iconic or Normal state. When the client changes one of these properties, it must use `Replace` mode to overwrite the entire property with new data; the window manager will retain no memory of the old value of the property. All fields of the property must be set to suitable values in a single `Replace` mode `ChangeProperty` request. This ensures that the full contents of the property will be available to a new window manager if the existing one crashes, if it is shut down and restarted, or if the session needs to be shut down and restarted by the session manager.

Convention

Clients writing or rewriting window manager properties must ensure that the entire content of each property remains valid at all times.

Some of these properties may contain the IDs of resources, such as windows or pixmaps. Clients should ensure that these resources exist for at least as long as the window on which the property resides.

If these properties are longer than expected, clients should ignore the remainder of the property. Extending these properties is reserved to the X Consortium; private extensions to them are forbidden. Private additional communication between clients and window managers should take place using separate properties. The only exception to this rule is the WM_PROTOCOLS property, which may be of arbitrary length and which may contain atoms representing private protocols (see WM_PROTOCOLS Property).

The next sections describe each of the properties the clients need to set, in turn. They are summarized in the table in Summary of Window Manager Property Types

WM_NAME Property

The WM_NAME property is an uninterpreted string that the client wants the window manager to display in association with the window (for example, in a window headline bar).

The encoding used for this string (and all other uninterpreted string properties) is implied by the type of the property. The type atoms to be used for this purpose are described in TEXT Properties.

Window managers are expected to make an effort to display this information. Simply ignoring WM_NAME is not acceptable behavior. Clients can assume that at least the first part of this string is visible to the user and that if the information is not visible to the user, it is because the user has taken an explicit action to make it invisible.

On the other hand, there is no guarantee that the user can see the WM_NAME string even if the window manager supports window headlines. The user may have placed the headline off-screen or have covered it by other windows. WM_NAME should not be used for application-critical information or to announce asynchronous changes of an application's state that require timely user response. The expected uses are to permit the user to identify one of a number of instances of the same client and to provide the user with noncritical state information.

Even window managers that support headline bars will place some limit on the length of the WM_NAME string that can be visible; brevity here will pay dividends.

WM_ICON_NAME Property

The WM_ICON_NAME property is an uninterpreted string that the client wants to be displayed in association with the window when it is iconified (for example, in an icon label). In other respects, including the type, it is similar to WM_NAME. For obvious geometric reasons, fewer characters will normally be visible in WM_ICON_NAME than WM_NAME.

Clients should not attempt to display this string in their icon pixmaps or windows; rather, they should rely on the window manager to do so.

WM_NORMAL_HINTS Property

The type of the WM_NORMAL_HINTS property is WM_SIZE_HINTS. Its contents are as follows:

Field	**Type**	**Comments**
flags	CARD32	(see the next table)
pad	4*CARD32	For backwards compatibility
min_width	INT32	If missing, assume base_width
min_height	INT32	If missing, assume base_height
max_width	INT32	
max_height	INT32	
width_inc	INT32	
height_inc	INT32	
min_aspect	(INT32,INT32)	
max_aspect	(INT32,INT32)	
base_width	INT32	If missing, assume min_width
base_height	INT32	If missing, assume min_height
win_gravity	INT32	If missing, assume `NorthWest`

The WM_SIZE_HINTS.flags bit definitions are as follows:

Name	**Value**	**Field**
`USPosition`	1	User-specified x, y
`USSize`	2	User-specified width, height
`PPosition`	4	Program-specified position
`PSize`	8	Program-specified size
`PMinSize`	16	Program-specified minimum size
`PMaxSize`	32	Program-specified maximum size
`PResizeInc`	64	Program-specified resize increments

Name	**Value**	**Field**
`PAspect`	128	Program-specified min and max aspect ratios
`PBaseSize`	256	Program-specified base size
`PWinGravity`	512	Program-specified window gravity

To indicate that the size and position of the window (when a transition from the Withdrawn state occurs) was specified by the user, the client should set the `USPosition` and `USSize` flags, which allow a window manager to know that the user specifically asked where the window should be placed or how the window should be sized and that further interaction is superfluous. To indicate that it was specified by the client without any user involvement, the client should set `PPosition` and `PSize`.

The size specifiers refer to the width and height of the client's window excluding borders.

The win_gravity may be any of the values specified for WINGRAVITY in the core protocol except for `Unmap`: `NorthWest` (1), `North` (2), `NorthEast` (3), `West` (4), `Center` (5), `East` (6), `SouthWest` (7), `South` (8), and `SouthEast` (9). It specifies how and whether the client window wants to be shifted to make room for the window manager frame.

If the win_gravity is `Static`, the window manager frame is positioned so that the inside border of the client window inside the frame is in the same position on the screen as it was when the client requested the transition from Withdrawn state. Other values of win_gravity specify a window reference point. For `NorthWest`, `NorthEast`, `SouthWest`, and `SouthEast` the reference point is the specified outer corner of the window (on the outside border edge). For `North`, `South`, `East` and `West` the reference point is the center of the specified outer edge of the window border. For `Center` the reference point is the center of the window. The reference point of the window manager frame is placed at the location on the screen where the reference point of the client window was when the client requested the transition from Withdrawn state.

The min_width and min_height elements specify the minimum size that the window can be for the client to be useful. The max_width and max_height elements specify the maximum size. The base_width and base_height elements in conjunction with width_inc and height_inc define an arithmetic progression of preferred window widths and heights for non-negative integers *i* and *j*:

```
width = base_width + ( i x width_inc )
```

```
height = base_height + ( j x height_inc )
```

Window managers are encouraged to use *i* and *j* instead of width and height in reporting window sizes to users. If a base size is not provided, the minimum size is to be used in its place and vice versa.

The min_aspect and max_aspect fields are fractions with the numerator first and the denominator second, and they allow a client to specify the range of aspect ratios it prefers. Window managers that honor aspect ratios should take into account the base size in determining the preferred window size. If a base size is provided along with the aspect ratio fields, the base size should be subtracted from the window size prior to checking that the aspect ratio falls in range. If a base size is not provided, nothing should be subtracted from the window size. (The minimum size is not to be used in place of the base size for this purpose.)

WM_HINTS Property

The WM_HINTS property (whose type is WM_HINTS) is used to communicate to the window manager. It conveys the information the window manager needs other than the window geometry, which is available from the window itself; the constraints on that geometry, which is available from the WM_NORMAL_HINTS structure; and various strings, which need separate properties, such as WM_NAME. The contents of the properties are as follows:

Field	**Type**	**Comments**
flags	CARD32	(see the next table)
input	CARD32	The client's input model
initial_state	CARD32	The state when first mapped
icon_pixmap	PIXMAP	The pixmap for the icon image
icon_window	WINDOW	The window for the icon image
icon_x	INT32	The icon location
icon_y	INT32	
icon_mask	PIXMAP	The mask for the icon shape
window_group	WINDOW	The ID of the group leader window

The WM_HINTS.flags bit definitions are as follows:

Name	**Value**	**Field**
`InputHint`	1	input
`StateHint`	2	initial_state
`IconPixmapHint`	4	icon_pixmap
`IconWindowHint`	8	icon_window
`IconPositionHint`	16	icon_x & icon_y
`IconMaskHint`	32	icon_mask
`WindowGroupHint`	64	window_group
`MessageHint`	128	(this bit is obsolete)
`UrgencyHint`	256	urgency

Window managers are free to assume convenient values for all fields of the WM_HINTS property if a window is mapped without one.

The input field is used to communicate to the window manager the input focus model used by the client (see Input Focus).

Clients with the Globally Active and No Input models should set the input flag to `False`. Clients with the Passive and Locally Active models should set the input flag to `True`.

From the client's point of view, the window manager will regard the client's top-level window as being in one of three states:

- Normal
- Iconic
- Withdrawn

The semantics of these states are described in Changing Window State. Newly created windows start in the Withdrawn state. Transitions between states happen when a top-level window is mapped and unmapped and when the window manager receives certain messages.

The value of the initial_state field determines the state the client wishes to be in at the time the top-level window is mapped from the Withdrawn state, as shown in the following table:

State	Value	Comments
NormalState	1	The window is visible
IconicState	3	The icon is visible

The icon_pixmap field may specify a pixmap to be used as an icon. This pixmap should be:

- One of the sizes specified in the WM_ICON_SIZE property on the root if it exists (see WM_ICON_SIZE Property).
- 1-bit deep. The window manager will select, through the defaults database, suitable background (for the 0 bits) and foreground (for the 1 bits) colors. These defaults can, of course, specify different colors for the icons of different clients.

The icon_mask specifies which pixels of the icon_pixmap should be used as the icon, allowing for icons to appear nonrectangular.

The icon_window field is the ID of a window the client wants used as its icon. Most, but not all, window managers will support icon windows. Those that do not are likely to have a user interface in which small windows that behave like icons are completely inappropriate. Clients should not attempt to remedy the omission by working around it.

Clients that need more capabilities from the icons than a simple 2-color bitmap should use icon windows. Rules for clients that do are set out in Icons.

The (icon_x,icon_y) coordinate is a hint to the window manager as to where it should position the icon. The policies of the window manager control the positioning of icons, so clients should not depend on attention being paid to this hint.

The window_group field lets the client specify that this window belongs to a group of windows. An example is a single client manipulating multiple children of the root window.

Conventions

- The window_group field should be set to the ID of the group leader. The window group leader may be a window that exists only for that purpose; a placeholder group leader of this kind would never be mapped either by the client or by the window manager.

- The properties of the window group leader are those for the group as a whole (for example, the icon to be shown when the entire group is iconified).

Window managers may provide facilities for manipulating the group as a whole. Clients, at present, have no way to operate on the group as a whole.

The messages bit, if set in the flags field, indicates that the client is using an obsolete window manager communication protocol, [1] rather than the WM_PROTOCOLS mechanism of WM_PROTOCOLS Property

The `UrgencyHint` flag, if set in the flags field, indicates that the client deems the window contents to be urgent, requiring the timely response of the user. The window manager must make some effort to draw the user's attention to this window while this flag is set. The window manager must also monitor the state of this flag for the entire time the window is in the Normal or Iconic state and must take appropriate action when the state of the flag changes. The flag is otherwise independent of the window's state; in particular, the window manager is not required to deiconify the window if the client sets the flag on an Iconic window. Clients must provide some means by which the user can cause the `UrgencyHint` flag to be set to zero or the window to be withdrawn. The user's action can either mitigate the actual condition that made the window urgent, or it can merely shut off the alarm.

Rationale

This mechanism is useful for alarm dialog boxes or reminder windows, in cases where mapping the window is not enough (e.g., in the presence of multi-workspace or virtual desktop window managers), and where using an override-redirect window is too intrusive. For example, the window manager may attract attention to an urgent window by adding an indicator to its title bar or its icon. Window managers may also take additional action for a window that is newly urgent, such as by flashing its icon (if the window is iconic) or by raising it to the top of the stack.

WM_CLASS Property

The WM_CLASS property (of type STRING without control characters) contains two consecutive null-terminated strings. These specify the Instance and Class names to be used by both the client and the window manager for looking up resources for the application or as identifying information. This property must be present when the window leaves the Withdrawn state and may be changed only while the window is in the Withdrawn state. Window managers may examine the property only when they start up and when the window leaves the Withdrawn state, but there should be no need for a client to change its state dynamically.

The two strings, respectively, are:

- A string that names the particular instance of the application to which the client that owns this window belongs. Resources that are specified by instance name override any resources that are specified by class name. Instance names can be

---

[1] This obsolete protocol was described in the July 27, 1988, draft of the ICCCM. Windows using it can also be detected because their WM_HINTS properties are 4 bytes longer than expected. Window managers are free to support clients using the obsolete protocol in a backwards compatibility mode.

specified by the user in an operating-system specific manner. On POSIX-conformant systems, the following conventions are used:

  - If "-name NAME" is given on the command line, NAME is used as the instance name.
  - Otherwise, if the environment variable RESOURCE_NAME is set, its value will be used as the instance name.
  - Otherwise, the trailing part of the name used to invoke the program (argv[0] stripped of any directory names) is used as the instance name.

- A string that names the general class of applications to which the client that owns this window belongs. Resources that are specified by class apply to all applications that have the same class name. Class names are specified by the application writer. Examples of commonly used class names include: "Emacs", "XTerm", "XClock", "XLoad", and so on.

Note that WM_CLASS strings are null-terminated and, thus, differ from the general conventions that STRING properties are null-separated. This inconsistency is necessary for backwards compatibility.

WM_TRANSIENT_FOR Property

The WM_TRANSIENT_FOR property (of type WINDOW) contains the ID of another top-level window. The implication is that this window is a pop-up on behalf of the named window, and window managers may decide not to decorate transient windows or may treat them differently in other ways. In particular, window managers should present newly mapped WM_TRANSIENT_FOR windows without requiring any user interaction, even if mapping top-level windows normally does require interaction. Dialogue boxes, for example, are an example of windows that should have WM_TRANSIENT_FOR set.

It is important not to confuse WM_TRANSIENT_FOR with override-redirect. WM_TRANSIENT_FOR should be used in those cases where the pointer is not grabbed while the window is mapped (in other words, if other windows are allowed to be active while the transient is up). If other windows must be prevented from processing input (for example, when implementing pop-up menus), use override-redirect and grab the pointer while the window is mapped.

WM_PROTOCOLS Property

The WM_PROTOCOLS property (of type ATOM) is a list of atoms. Each atom identifies a communication protocol between the client and the window manager in which the client is willing to participate. Atoms can identify both standard protocols and private protocols specific to individual window managers.

All the protocols in which a client can volunteer to take part involve the window manager sending the client a `ClientMessage` event and the client taking appropriate action. For details of the contents of the event, see ClientMessage Events In each case, the protocol transactions are initiated by the window manager.

The WM_PROTOCOLS property is not required. If it is not present, the client does not want to participate in any window manager protocols.

The X Consortium will maintain a registry of protocols to avoid collisions in the name space. The following table lists the protocols that have been defined to date.

Protocol	**Section**	**Purpose**
WM_TAKE_FOCUS	Input Focus	Assignment of input focus
WM_SAVE_YOURSELF	Appendix C	Save client state request (deprecated)
WM_DELETE_WINDOW	Window Deletion	Request to delete top-level window

It is expected that this table will grow over time.

WM_COLORMAP_WINDOWS Property

The WM_COLORMAP_WINDOWS property (of type WINDOW) on a top-level window is a list of the IDs of windows that may need colormaps installed that differ from the colormap of the top-level window. The window manager will watch this list of windows for changes in their colormap attributes. The top-level window is always (implicitly or explicitly) on the watch list. For the details of this mechanism, see Colormaps

WM_CLIENT_MACHINE Property

The client should set the WM_CLIENT_MACHINE property (of one of the TEXT types) to a string that forms the name of the machine running the client as seen from the machine running the server.

Window Manager Properties

The properties that were described in the previous section are those that the client is responsible for maintaining on its top-level windows. This section describes the properties that the window manager places on client's top-level windows and on the root.

WM_STATE Property

The window manager will place a WM_STATE property (of type WM_STATE) on each top-level client window that is not in the Withdrawn state. Top-level windows in the Withdrawn state may or may not have the WM_STATE property. Once the top-level window has been withdrawn, the client may re-use it for another purpose. Clients that do so should remove the WM_STATE property if it is still present.

Some clients (such as `xprop`) will ask the user to click over a window on which the program is to operate. Typically, the intent is for this to be a top-level window. To find a top-level window, clients should search the window hierarchy beneath the selected location for a window with the WM_STATE property. This search must be recursive in order to cover all window manager reparenting possibilities. If no window with a WM_STATE property is found, it is recommended that programs use a mapped child-of-root window if one is present beneath the selected location.

The contents of the WM_STATE property are defined as follows:

Field	Type	Comments
state	CARD32	(see the next table)
icon	WINDOW	ID of icon window

The following table lists the WM_STATE.state values:

State	Value
WithdrawnState	0
NormalState	1
IconicState	3

Adding other fields to this property is reserved to the X Consortium. Values for the state field other than those defined in the above table are reserved for use by the X Consortium.

The state field describes the window manager's idea of the state the window is in, which may not match the client's idea as expressed in the initial_state field of the WM_HINTS property (for example, if the user has asked the window manager to iconify the window). If it is `NormalState`, the window manager believes the client should be animating its window. If it is `IconicState`, the client should animate its icon window. In either state, clients should be prepared to handle exposure events from either window.

When the window is withdrawn, the window manager will either change the state field's value to `WithdrawnState` or it will remove the WM_STATE property entirely.

The icon field should contain the window ID of the window that the window manager uses as the icon for the window on which this property is set. If no such window exists, the icon field should be `None`. Note that this window could be but is not necessarily the same window as the icon window that the client may have specified in its WM_HINTS property. The WM_STATE icon may be a window that the window manager has supplied and that contains the client's icon pixmap, or it may be an ancestor of the client's icon window.

WM_ICON_SIZE Property

A window manager that wishes to place constraints on the sizes of icon pixmaps and/or windows should place a property called WM_ICON_SIZE on the root. The contents of this property are listed in the following table.

Field	Type	Comments
min_width	CARD32	The data for the icon size series
min_height	CARD32	
max_width	CARD32	
max_height	CARD32	
width_inc	CARD32	
height_inc	CARD32	

For more details see section 14.1.12 in *Xlib - C Language X Interface*.

Changing Window State

From the client's point of view, the window manager will regard each of the client's top-level windows as being in one of three states, whose semantics are as follows:

- `NormalState` - The client's top-level window is viewable.
- `IconicState` - The client's top-level window is iconic (whatever that means for this window manager). The client can assume that its top-level window is not viewable, its icon_window (if any) will be viewable and, failing that, its icon_pixmap (if any) or its WM_ICON_NAME will be displayed.
- `WithdrawnState` - Neither the client's top-level window nor its icon is visible.

In fact, the window manager may implement states with semantics other than those described above. For example, a window manager might implement a concept of an "inactive" state in which an infrequently used client's window would be represented as a string in a menu. But this state is invisible to the client, which would see itself merely as being in the Iconic state.

Newly created top-level windows are in the Withdrawn state. Once the window has been provided with suitable properties, the client is free to change its state as follows:

- Withdrawn -> Normal - The client should map the window with WM_HINTS.initial_state being `NormalState`.
- Withdrawn -> Iconic - The client should map the window with WM_HINTS.initial_state being `IconicState`.
- Normal -> Iconic - The client should send a `ClientMessage` event as described later in this section.
- Normal -> Withdrawn - The client should unmap the window and follow it with a synthetic `UnmapNotify` event as described later in this section.
- Iconic -> Normal - The client should map the window. The contents of WM_HINTS.initial_state are irrelevant in this case.
- Iconic -> Withdrawn - The client should unmap the window and follow it with a synthetic `UnmapNotify` event as described later in this section.

Only the client can effect a transition into or out of the Withdrawn state. Once a client's window has left the Withdrawn state, the window will be mapped if it is in the Normal state and the window will be unmapped if it is in the Iconic state. Reparenting window managers must unmap the client's window when it is in the Iconic state, even if an ancestor window being unmapped renders the client's window unviewable. Conversely, if a reparenting window manager renders the client's window unviewable by unmapping an ancestor, the client's window is by definition in the Iconic state and must also be unmapped.

Advice to Implementors

Clients can select for `StructureNotify` on their top-level windows to track transitions between Normal and Iconic states. Receipt of a `MapNotify` event will indicate a transition to the Normal state, and receipt of an `UnmapNotify` event will indicate a transition to the Iconic state.

When changing the state of the window to Withdrawn, the client must (in addition to unmapping the window) send a synthetic `UnmapNotify` event by using a `SendEvent` request with the following arguments:

Argument	**Value**
destination	The root
propogate	**False**
event-mask	(**SubstructureRedirect|SubstructureNotify**)
event: an `UnmapNotify` with:	
event:	The root
window:	The window itself
from-configure:	**False**

Rationale

The reason for requiring the client to send a synthetic `UnmapNotify` event is to ensure that the window manager gets some notification of the client's desire to change state, even though the window may already be unmapped when the desire is expressed.

Advice to Implementors

For compatibility with obsolete clients, window managers should trigger the transition to the Withdrawn state on the real `UnmapNotify` rather than waiting for the synthetic one. They should also trigger the transition if they receive a synthetic `UnmapNotify` on a window for which they have not yet received a real `UnmapNotify`.

When a client withdraws a window, the window manager will then update or remove the WM_STATE property as described in WM_STATE Property. Clients that want to re-use a client window (e.g., by mapping it again or reparenting it elsewhere) after withdrawing it must wait for the withdrawal to be complete before proceeding. The preferred method for doing this is for clients to wait for the window manager to update or remove the WM_STATE property. [2]

If the transition is from the Normal to the Iconic state, the client should send a `ClientMessage` event to the root with:

- Window == the window to be iconified
- Type [3] == the atom WM_CHANGE_STATE
- Format == 32

---

[2] Earlier versions of these conventions prohibited clients from reading the WM_STATE property. Clients operating under the earlier conventions used the technique of tracking `ReparentNotify` events to wait for the top-level window to be reparented back to the root window. This is still a valid technique; however, it works only for reparenting window managers, and the WM_STATE technique is to be preferred.

[3] The type field of the `ClientMessage` event (called the message_type field by Xlib) should not be confused with the code field of the event itself, which will have the value 33 (`ClientMessage`).

- Data[0] == IconicState

Rationale

The format of this `ClientMessage` event does not match the format of `ClientMessages` in ClientMessage Events. This is because they are sent by the window manager to clients, and this message is sent by clients to the window manager.

Other values of data[0] are reserved for future extensions to these conventions. The parameters of the `SendEvent` request should be those described for the synthetic `UnmapNotify` event.

Advice to Implementors

Clients can also select for `VisibilityChange` events on their top-level or icon windows. They will then receive a `VisibilityNotify` (state==FullyObscured) event when the window concerned becomes completely obscured even though mapped (and thus, perhaps a waste of time to update) and a `VisibilityNotify` (state!=FullyObscured) event when it becomes even partly viewable.

Advice to Implementors

When a window makes a transition from the Normal state to either the Iconic or the Withdrawn state, clients should be aware that the window manager may make transients for this window inaccessible. Clients should not rely on transient windows being available to the user when the transient owner window is not in the Normal state. When withdrawing a window, clients are advised to withdraw transients for the window.

Configuring the Window

Clients can resize and reposition their top-level windows by using the `ConfigureWindow` request. The attributes of the window that can be altered with this request are as follows:

- The [x,y] location of the window's upper left-outer corner
- The [width,height] of the inner region of the window (excluding borders)
- The border width of the window
- The window's position in the stack

The coordinate system in which the location is expressed is that of the root (irrespective of any reparenting that may have occurred). The border width to be used and win_gravity position hint to be used are those most recently requested by the client. Client configure requests are interpreted by the window manager in the same manner as the initial window geometry mapped from the Withdrawn state, as described in WM_NORMAL_HINTS Property Clients must be aware that there is no guarantee that the window manager will allocate them the requested size or location and must be prepared to deal with any size and location. If the window manager decides to respond to a `ConfigureRequest` request by:

- Not changing the size, location, border width, or stacking order of the window at all.

  A client will receive a synthetic `ConfigureNotify` event that describes the (unchanged) geometry of the window. The (x,y) coordinates will be in the root coordinate system, adjusted for the border width the client requested, irrespective of any reparenting that has taken place. The border_width will be the border width the client requested. The client will not receive a real `ConfigureNotify` event because no change has actually taken place.

- Moving or restacking the window without resizing it or changing its border width.

  A client will receive a synthetic `ConfigureNotify` event following the change that describes the new geometry of the window. The event's (x,y) coordinates will be in the root coordinate system adjusted for the border width the client requested. The border_width will be the border width the client requested. The client may not receive a real `ConfigureNotify` event that describes this change because the window manager may have reparented the top-level window. If the client does receive a real event, the synthetic event will follow the real one.

- Resizing the window or changing its border width (regardless of whether the window was also moved or restacked).

  A client that has selected for `StructureNotify` events will receive a real `ConfigureNotify` event. Note that the coordinates in this event are relative to the parent, which may not be the root if the window has been reparented. The coordinates will reflect the actual border width of the window (which the window manager may have changed). The `TranslateCoordinates` request can be used to convert the coordinates if required.

The general rule is that coordinates in real `ConfigureNotify` events are in the parent's space; in synthetic events, they are in the root space.

Advice to Implementors

Clients cannot distinguish between the case where a top-level window is resized and moved from the case where the window is resized but not moved, since a real `ConfigureNotify` event will be received in both cases. Clients that are concerned with keeping track of the absolute position of a top-level window should keep a piece of state indicating whether they are certain of its position. Upon receipt of a real `ConfigureNotify` event on the top-level window, the client should note that the position is unknown. Upon receipt of a synthetic `ConfigureNotify` event, the client should note the position as known, using the position in this event. If the client receives a `KeyPress`, `KeyRelease`, `ButtonPress`, `ButtonRelease`, `MotionNotify`, `EnterNotify` or `LeaveNotify` event on the window (or on any descendant), the client can deduce the top-level window's position from the difference between the (event-x, event-y) and (root-x, root-y) coordinates in these events. Only when the position is unknown does the client need to use the `TranslateCoordinates` request to find the position of a top-level window.

Clients should be aware that their borders may not be visible. Window managers are free to use reparenting techniques to decorate client's top-level windows with borders containing titles, controls, and other details to maintain a consistent look-and-feel. If they do, they are likely to override the client's attempts to set the border width and set it to zero. Clients, therefore, should not depend on the top-level window's border being visible or use it to display any critical information. Other window managers will allow the top-level windows border to be visible.

Convention

Clients should set the desired value of the border-width attribute on all `ConfigureWindow` requests to avoid a race condition.

Clients that change their position in the stack must be aware that they may have been reparented, which means that windows that used to be siblings no longer are. Using a nonsibling as the sibling parameter on a `ConfigureWindow` request will cause an error.

Convention

Clients that use a `ConfigureWindow` request to request a change in their position in the stack should do so using `None` in the sibling field.

Clients that must position themselves in the stack relative to some window that was originally a sibling must do the `ConfigureWindow` request (in case they are running under a nonreparenting window manager), be prepared to deal with a resulting error, and then follow with a synthetic `ConfigureRequest` event by invoking a `SendEvent` request with the following arguments:

Argument	Value
destination	The root
propogate	**False**
event-mask	(**SubstructureRedirect|SubstructureNotify**)
event: an `ConfigureRequest` with:	
event:	The root
window:	The window itself
...	Other parameters from the **ConfigureWindow** request

Window managers are in any case free to position windows in the stack as they see fit, and so clients should not rely on receiving the stacking order they have requested. Clients should ignore the above-sibling field of both real and synthetic `ConfigureNotify` events received on their top-level windows because this field may not contain useful information.

Changing Window Attributes

The attributes that may be supplied when a window is created may be changed by using the `ChangeWindowAttributes` request. The window attributes are listed in the following table:

Attribute	Private to Client
Background pixmap	Yes

Attribute	Private to Client
Background pixel	Yes
Border pixmap	Yes
Border pixel	Yes
Bit gravity	Yes
Window gravity	No
Backing-store hint	Yes
Save-under hint	No
Event Mask	No
Do-not-propagate mask	Yes
Override-redirect flag	No
Colormap	Yes
Cursor	Yes

Most attributes are private to the client and will never be interfered with by the window manager. For the attributes that are not private to the client:

- The window manager is free to override the window gravity; a reparenting window manager may want to set the top-level window's window gravity for its own purposes.

- Clients are free to set the save-under hint on their top-level windows, but they must be aware that the hint may be overridden by the window manager.

- Windows, in effect, have per-client event masks, and so, clients may select for whatever events are convenient irrespective of any events the window manager is selecting for. There are some events for which only one client at a time may select, but the window manager should not select for them on any of the client's windows.

- Clients can set override-redirect on top-level windows but are encouraged not to do so except as described in Pop-up Windows. and Redirecting Requests.

Input Focus

There are four models of input handling:

- No Input - The client never expects keyboard input. An example would be `xload` or another output-only client.

- Passive Input - The client expects keyboard input but never explicitly sets the input focus. An example would be a simple client with no subwindows, which will accept input in `PointerRoot` mode or when the window manager sets the input focus to its top-level window (in click-to-type mode).

- Locally Active Input - The client expects keyboard input and explicitly sets the input focus, but it only does so when one of its windows already has the focus. An example would be a client with subwindows defining various data entry fields that uses Next and Prev keys to move the input focus between the fields. It does so when its top-level window has acquired the focus in `PointerRoot` mode or when the window manager sets the input focus to its top-level window (in click-to-type mode).

- Globally Active Input - The client expects keyboard input and explicitly sets the input focus, even when it is in windows the client does not own. An example would be a client with a scroll bar that wants to allow users to scroll the window without disturbing the input focus even if it is in some other window. It wants to acquire the input focus when the user clicks in the scrolled region but not when the user clicks in the scroll bar itself. Thus, it wants to prevent the window manager from setting the input focus to any of its windows.

The four input models and the corresponding values of the input field and the presence or absence of the WM_TAKE_FOCUS atom in the WM_PROTOCOLS property are listed in the following table:

Input Model	**Input Field**	**WM_TAKE_FOCUS**
No Input	**False**	Absent
Passive	**True**	Absent
Locally Active	**True**	Present
Globally Active	**False**	Present

Passive and Locally Active clients set the input field of WM_HINTS to `True`, which indicates that they require window manager assistance in acquiring the input focus. No Input and Globally Active clients set the input field to `False`, which requests that the window manager not set the input focus to their top-level window.

Clients that use a `SetInputFocus` request must set the time field to the timestamp of the event that caused them to make the attempt. This cannot be a `FocusIn` event because they do not have timestamps. Clients may also acquire the focus without a corresponding `EnterNotify`. Note that clients must not use `CurrentTime` in the time field.

Clients using the Globally Active model can only use a `SetInputFocus` request to acquire the input focus when they do not already have it on receipt of one of the following events:

- `ButtonPress`
- `ButtonRelease`
- Passive-grabbed `KeyPress`
- Passive-grabbed `KeyRelease`

In general, clients should avoid using passive-grabbed key events for this purpose, except when they are unavoidable (as, for example, a selection tool that establishes a passive grab on the keys that cut, copy, or paste).

The method by which the user commands the window manager to set the focus to a window is up to the window manager. For example, clients cannot determine whether they will see the click that transfers the focus.

Windows with the atom WM_TAKE_FOCUS in their WM_PROTOCOLS property may receive a `ClientMessage` event from the window manager (as described in

ClientMessage Events.) with WM_TAKE_FOCUS in its data[0] field and a valid timestamp (i.e., not `CurrentTime`) in its data[1] field. If they want the focus, they should respond with a `SetInputFocus` request with its window field set to the window of theirs that last had the input focus or to their default input window, and the time field set to the timestamp in the message. For further information, see Input Focus

A client could receive WM_TAKE_FOCUS when opening from an icon or when the user has clicked outside the top-level window in an area that indicates to the window manager that it should assign the focus (for example, clicking in the headline bar can be used to assign the focus).

The goal is to support window managers that want to assign the input focus to a top-level window in such a way that the top-level window either can assign it to one of its subwindows or can decline the offer of the focus. For example, a clock or a text editor with no currently open frames might not want to take focus even though the window manager generally believes that clients should take the input focus after being deiconified or raised.

Clients that set the input focus need to decide a value for the revert-to field of the `SetInputFocus` request. This determines the behavior of the input focus if the window the focus has been set to becomes not viewable. The value can be any of the following:

- `Parent` - In general, clients should use this value when assigning focus to one of their subwindows. Unmapping the subwindow will cause focus to revert to the parent, which is probably what you want.
- `PointerRoot` - Using this value with a click-to-type focus management policy leads to race conditions because the window becoming unviewable may coincide with the window manager deciding to move the focus elsewhere.
- `None` - Using this value causes problems if the window manager reparents the window, as most window managers will, and then crashes. The input focus will be `None`, and there will probably be no way to change it.

Note that neither `PointerRoot` nor `None` is really safe to use.

Convention

Clients that invoke a `SetInputFocus` request should set the revert-to argument to `Parent`.

A convention is also required for clients that want to give up the input focus. There is no safe value set for them to set the input focus to; therefore, they should ignore input material.

Convention

Clients should not give up the input focus of their own volition. They should ignore input that they receive instead.

Colormaps

The window manager is responsible for installing and uninstalling colormaps on behalf of clients with top-level windows that the window manager manages.

Clients provide the window manager with hints as to which colormaps to install and uninstall. Clients must not install or uninstall colormaps themselves (except under the circumstances noted below). When a client's top-level window gets the colormap focus (as a result of whatever colormap focus policy is implemented by the window manager), the window manager will ensure that one or more of the client's colormaps are installed.

Clients whose top-level windows and subwindows all use the same colormap should set its ID in the colormap field of the top-level window's attributes. They should not set a WM_COLORMAP_WINDOWS property on the top-level window. If they want to change the colormap, they should change the top-level window's colormap attribute. The window manager will track changes to the window's colormap attribute and install colormaps as appropriate.

Clients that create windows can use the value `CopyFromParent` to inherit their parent's colormap. Window managers will ensure that the root window's colormap field contains a colormap that is suitable for clients to inherit. In particular, the colormap will provide distinguishable colors for `BlackPixel` and `WhitePixel`.

Top-level windows that have subwindows or override-redirect pop-up windows whose colormap requirements differ from the top-level window should have a WM_COLORMAP_WINDOWS property. This property contains a list of IDs for windows whose colormaps the window manager should attempt to have installed when, in the course of its individual colormap focus policy, it assigns the colormap focus to the top-level window (see WM_COLORMAP_WINDOWS Property). The list is ordered by the importance to the client of having the colormaps installed. The window manager will track changes to this property and will track changes to the colormap attribute of the windows in the property.

If the relative importance of colormaps changes, the client should update the WM_COLORMAP_WINDOWS property to reflect the new ordering. If the top-level window does not appear in the list, the window manager will assume it to be of higher priority than any window in the list.

WM_TRANSIENT_FOR windows can either have their own WM_COLORMAP_WINDOWS property or appear in the property of the window they are transient for, as appropriate.

Rationale

An alternative design was considered for how clients should hint to the window manager about their colormap requirements. This alternative design specified a list of colormaps instead of a list of windows. The current design, a list of windows, was chosen for two reasons. First, it allows window managers to find the visuals of the colormaps, thus permitting visual-dependent colormap installation policies. Second, it allows window managers to select for `VisibilityChange` events on the windows concerned and to ensure that colormaps are only installed if the windows that need them are visible. The alternative design allows for neither of these policies.

Advice to Implementors

Clients should be aware of the min-installed-maps and max-installed-maps fields of the connection setup information, and the ef-

> fect that the minimum value has on the "required list" defined by the Protocol in the description of the `InstallColormap` request. Briefly, the min-installed-maps most recently installed maps are guaranteed to be installed. This value is often one; clients needing multiple colormaps should beware.

Whenever possible, clients should use the mechanisms described above and let the window manager handle colormap installation. However, clients are permitted to perform colormap installation on their own while they have the pointer grabbed. A client performing colormap installation must notify the window manager prior to the first installation. When the client has finished its colormap installation, it must also notify the window manager. The client notifies the window manager by issuing a `SendEvent` request with the following arguments:

Argument	**Value**
destination	The root window of the screen on which the colormap is installed
propogate	**False**
event-mask	**ColormapChange**
event: an `ClientMessage` with:	
window:	The root window, as above
type:	WM_COLORMAP_NOTIFY
format	32
data[0]	the timestampe of the event that caused the client to start or stop installing colormaps
data[1]	1 if the client is starting colormap installation, 0 if the client is finished with colormap installation
data[2]	reserved, must be zero
data[3]	reserved, must be zero
data[4]	reserved, must be zero

This feature was introduced in version 2.0 of this document, and there will be a significant period of time before all window managers can be expected to implement this feature. Before using this feature, clients must check the compliance level of the window manager (using the mechanism described in Communication with the Window Manager by Means of Selections) to verify that it supports this feature. This is necessary to prevent colormap installation conflicts between clients and older window managers.

Window managers should refrain from installing colormaps while a client has requested control of colormap installation. The window manager should continue to track the set of installed colormaps so that it can reinstate its colormap focus policy when the client has finished colormap installation.

This technique has race conditions that may result in the colormaps continuing to be installed even after a client has issued its notification message. For example, the window manager may have issued some `InstallColormap` requests that are not executed until after the client's `SendEvent` and `InstallColormap` requests, thus uninstalling the client's colormaps. If this occurs while the client still has the pointer grabbed and before the client has issued the "finished" message, the client may reinstall the desired colormaps.

Advice to Implementors

Clients are expected to use this mechanism for things such as pop-up windows and for animations that use override-redirect windows.

If a client fails to issue the "finished" message, the window manager may be left in a state where its colormap installation policy is suspended. Window manager implementors may want to implement a feature that resets colormap installation policy in response to a command from the user.

Icons

A client can hint to the window manager about the desired appearance of its icon by setting:

- A string in WM_ICON_NAME.
- All clients should do this because it provides a fallback for window managers whose ideas about icons differ widely from those of the client.

  A `Pixmap` into the icon_pixmap field of the WM_HINTS property and possibly another into the icon_mask field.
- The window manager is expected to display the pixmap masked by the mask. The pixmap should be one of the sizes found in the WM_ICON_SIZE property on the root. If this property is not found, the window manager is unlikely to display icon pixmaps. Window managers usually will clip or tile pixmaps that do not match WM_ICON_SIZE.
- A window into the icon_window field of the WM_HINTS property.

  The window manager is expected to map that window whenever the client is in the Iconic state. In general, the size of the icon window should be one of those specified in WM_ICON_SIZE on the root, if it exists. Window managers are free to resize icon windows.

In the Iconic state, the window manager usually will ensure that:

- If the window's WM_HINTS.icon_window is set, the window it names is visible.
- If the window's WM_HINTS.icon_window is not set but the window's WM_HINTS.icon_pixmap is set, the pixmap it names is visible.
- Otherwise, the window's WM_ICON_NAME string is visible.

Clients should observe the following conventions about their icon windows:

Conventions

- The icon window should be an `InputOutput` child of the root.

- The icon window should be one of the sizes specified in the WM_ICON_SIZE property on the root.
- The icon window should use the root visual and default colormap for the screen in question.
- Clients should not map their icon windows.
- Clients should not unmap their icon windows.
- Clients should not configure their icon windows.
- Clients should not set override-redirect on their icon windows or select for `ResizeRedirect` events on them.
- Clients must not depend on being able to receive input events by means of their icon windows.
- Clients must not manipulate the borders of their icon windows.
- Clients must select for `Exposure` events on their icon window and repaint it when requested.

Window managers will differ as to whether they support input events to client's icon windows; most will allow the client to receive some subset of the keys and buttons.

Window managers will ignore any WM_NAME, WM_ICON_NAME, WM_NORMAL_HINTS, WM_HINTS, WM_CLASS, WM_TRANSIENT_FOR, WM_PROTOCOLS, WM_COLORMAP_WINDOWS, WM_COMMAND, or WM_CLIENT_MACHINE properties they find on icon windows.

Pop-up Windows

Clients that wish to pop up a window can do one of three things:

- They can create and map another normal top-level window, which will get decorated and managed as normal by the window manager. See the discussion of window groups that follows.
- If the window will be visible for a relatively short time and deserves a somewhat lighter treatment, they can set the WM_TRANSIENT_FOR property. They can expect less decoration but can set all the normal window manager properties on the window. An example would be a dialog box.
- If the window will be visible for a very short time and should not be decorated at all, the client can set override-redirect on the window. In general, this should be done only if the pointer is grabbed while the window is mapped. The window manager will never interfere with these windows, which should be used with caution. An example of an appropriate use is a pop-up menu.

Advice to Implementors

The user will not be able to move, resize, restack, or transfer the input focus to override-redirect windows, since the window manager is not managing them. If it is necessary for a client to receive keystrokes on an override-redirect window, either the client must grab the keyboard or the client must have another top-level window that is not

> override-redirect and that has selected the Locally Active or Globally Active focus model. The client may set the focus to the override-redirect window when the other window receives a WM_TAKE_FOCUS message or one of the events listed in Input Focus in the description of the Globally Active focus model.

Window managers are free to decide if WM_TRANSIENT_FOR windows should be iconified when the window they are transient for is. Clients displaying WM_TRANSIENT_FOR windows that have (or request to have) the window they are transient for iconified do not need to request that the same operation be performed on the WM_TRANSIENT_FOR window; the window manager will change its state if that is the policy it wishes to enforce.

Window Groups

A set of top-level windows that should be treated from the user's point of view as related (even though they may belong to a number of clients) should be linked together using the window_group field of the WM_HINTS structure.

One of the windows (that is, the one the others point to) will be the group leader and will carry the group as opposed to the individual properties. Window managers may treat the group leader differently from other windows in the group. For example, group leaders may have the full set of decorations, and other group members may have a restricted set.

It is not necessary that the client ever map the group leader; it may be a window that exists solely as a placeholder.

It is up to the window manager to determine the policy for treating the windows in a group. At present, there is no way for a client to request a group, as opposed to an individual, operation.

Client Responses to Window Manager Actions

The window manager performs a number of operations on client resources, primarily on their top-level windows. Clients must not try to fight this but may elect to receive notification of the window manager's operations.

Reparenting

Clients must be aware that some window managers will reparent their top-level windows so that a window that was created as a child of the root will be displayed as a child of some window belonging to the window manager. The effects that this reparenting will have on the client are as follows:

- The parent value returned by a `QueryTree` request will no longer be the value supplied to the `CreateWindow` request that created the reparented window. There should be no need for the client to be aware of the identity of the window to which the top-level window has been reparented. In particular, a client that wishes to create further top-level windows should continue to use the root as the parent for these new windows.
- The server will interpret the (x,y) coordinates in a `ConfigureWindow` request in the new parent's coordinate space. In fact, they usually will not be interpreted by

the server because a reparenting window manager usually will have intercepted these operations (see Redirection of Operations). Clients should use the root coordinate space for these requests (see Configuring the Window).

- `ConfigureWindow` requests that name a specific sibling window may fail because the window named, which used to be a sibling, no longer is after the reparenting operation (see Configuring the Window).
- The (x,y) coordinates returned by a `GetGeometry` request are in the parent's coordinate space and are thus not directly useful after a reparent operation.
- A background of `ParentRelative` will have unpredictable results.
- A cursor of `None` will have unpredictable results.

Clients that want to be notified when they are reparented can select for `StructureNotify` events on their top-level window. They will receive a `ReparentNotify` event if and when reparenting takes place. When a client withdraws a top-level window, the window manager will reparent it back to the root window if the window had been reparented elsewhere.

If the window manager reparents a client's window, the reparented window will be placed in the save-set of the parent window. This means that the reparented window will not be destroyed if the window manager terminates and will be remapped if it was unmapped. Note that this applies to all client windows the window manager reparents, including transient windows and client icon windows.

Redirection of Operations

Clients must be aware that some window managers will arrange for some client requests to be intercepted and redirected. Redirected requests are not executed; they result instead in events being sent to the window manager, which may decide to do nothing, to alter the arguments, or to perform the request on behalf of the client.

The possibility that a request may be redirected means that a client cannot assume that any redirectable request is actually performed when the request is issued or is actually performed at all. The requests that may be redirected are `MapWindow`, `ConfigureWindow`, and `CirculateWindow`.

Advice to Implementors

The following is incorrect because the `MapWindow` request may be intercepted and the `PolyLine` output made to an unmapped window:

```
MapWindow A
PolyLine A GC <point> <point> ...
```

The client must wait for an `Expose` event before drawing in the window. [4]

This next example incorrectly assumes that the `ConfigureWindow` request is actually executed with the arguments supplied:

---

[4] This is true even if the client set the backing-store attribute to `Always`. The backing-store attribute is a only a hint, and the server may stop maintaining backing store contents at any time.

```
ConfigureWindow width=N height=M
<output assuming window is N by M>
```

The client should select for `StructureNotify` on its window and monitor the window's size by tracking `ConfigureNotify` events.

Clients must be especially careful when attempting to set the focus to a window that they have just mapped. This sequence may result in an X protocol error:

```
MapWindow B
SetInputFocus B
```

If the `MapWindow` request has been intercepted, the window will still be unmapped, causing the `SetInputFocus` request to generate the error. The solution to this problem is for clients to select for `VisibilityChange` on the window and to delay the issuance of the `SetInputFocus` request until they have received a `VisibilityNotify` event indicating that the window is visible.

This technique does not guarantee correct operation. The user may have iconified the window by the time the `SetInputFocus` request reaches the server, still causing an error. Or the window manager may decide to map the window into Iconic state, in which case the window will not be visible. This will delay the generation of the `VisibilityNotify` event indefinitely. Clients must be prepared to handle these cases.

A window with the override-redirect bit set is immune from redirection, but the bit should be set on top-level windows only in cases where other windows should be prevented from processing input while the override-redirect window is mapped (see Pop-up Windows) and while responding to `ResizeRequest` events (see Redirecting Requests).

Clients that have no non-Withdrawn top-level windows and that map an override-redirect top-level window are taking over total responsibility for the state of the system. It is their responsibility to:

- Prevent any preexisting window manager from interfering with their activities
- Restore the status quo exactly after they unmap the window so that any preexisting window manager does not get confused

In effect, clients of this kind are acting as temporary window managers. Doing so is strongly discouraged because these clients will be unaware of the user interface policies the window manager is trying to maintain and because their user interface behavior is likely to conflict with that of less demanding clients.

Window Move

If the window manager moves a top-level window without changing its size, the client will receive a synthetic `ConfigureNotify` event following the move that describes the new location in terms of the root coordinate space. Clients must not respond to being moved by attempting to move themselves to a better location.

Any real `ConfigureNotify` event on a top-level window implies that the window's position on the root may have changed, even though the event reports that the window's position in its parent is unchanged because the window may have been reparented. Note that the coordinates in the event will not, in this case, be directly useful.

The window manager will send these events by using a `SendEvent` request with the following arguments:

Argument	**Value**
destination	The client's window
propagate	**False**
event-mask	**StructureNotify**

Window Resize

The client can elect to receive notification of being resized by selecting for `StructureNotify` events on its top-level windows. It will receive a `ConfigureNotify` event. The size information in the event will be correct, but the location will be in the parent window (which may not be the root).

The response of the client to being resized should be to accept the size it has been given and to do its best with it. Clients must not respond to being resized by attempting to resize themselves to a better size. If the size is impossible to work with, clients are free to request to change to the Iconic state.

Iconify and Deiconify

A top-level window that is not Withdrawn will be in the Normal state if it is mapped and in the Iconic state if it is unmapped. This will be true even if the window has been reparented; the window manager will unmap the window as well as its parent when switching to the Iconic state.

The client can elect to be notified of these state changes by selecting for `StructureNotify` events on the top-level window. It will receive a `UnmapNotify` event when it goes Iconic and a `MapNotify` event when it goes Normal.

Colormap Change

Clients that wish to be notified of their colormaps being installed or uninstalled should select for `ColormapNotify` events on their top-level windows and on any windows they have named in WM_COLORMAP_WINDOWS properties on their top-level windows. They will receive `ColormapNotify` events with the new field FALSE when the colormap for that window is installed or uninstalled.

Input Focus

Clients can request notification that they have the input focus by selecting for `FocusChange` events on their top-level windows; they will receive `FocusIn` and `FocusOut` events. Clients that need to set the input focus to one of their subwindows should not do so unless they have set WM_TAKE_FOCUS in their WM_PROTOCOLS property and have done one of the following:

- Set the input field of WM_HINTS to `True` and actually have the input focus in one of their top-level windows
- Set the input field of WM_HINTS to `False` and have received a suitable event as described in Input Focus.
- Have received a WM_TAKE_FOCUS message as described in Input Focus.

Clients should not warp the pointer in an attempt to transfer the focus; they should set the focus and leave the pointer alone. For further information, see The Pointer.

Once a client satisfies these conditions, it may transfer the focus to another of its windows by using the `SetInputFocus` request, which is defined as follows:

`SetInputFocus`

focus: WINDOW or **PointerRoot** or **None**

revert-to: { **Parent**, **PointerRoot**, **None** }

time: TIMESTAMP or CurrentTime

Conventions

- Clients that use a `SetInputFocus` request must set the time argument to the timestamp of the event that caused them to make the attempt. This cannot be a `FocusIn` event because they do not have timestamps. Clients may also acquire the focus without a corresponding `EnterNotify` event. Clients must not use `CurrentTime` for the time argument.
- Clients that use a `SetInputFocus` request to set the focus to one of their windows must set the revert-to field to `Parent`.

ClientMessage Events

There is no way for clients to prevent themselves being sent `ClientMessage` events.

Top-level windows with a WM_PROTOCOLS property may be sent `ClientMessage` events specific to the protocols named by the atoms in the property (see WM_PROTOCOLS Property). For all protocols, the `ClientMessage` events have the following:

- WM_PROTOCOLS as the type field
- Format 32
- The atom that names their protocol in the data[0] field
- A timestamp in their data[1] field

The remaining fields of the event, including the window field, are determined by the protocol.

These events will be sent by using a `SendEvent` request with the following arguments:

Argument	**Value**
destination	The client's window
propagate	**False**
event-mask	() empty
event	As specified by the protocol

Window Deletion

Clients, usually those with multiple top-level windows, whose server connection must survive the deletion of some of their top-level windows, should include the atom WM_DELETE_WINDOW in the WM_PROTOCOLS property on each such window. They will receive a `ClientMessage` event as described above whose data[0] field is WM_DELETE_WINDOW.

Clients receiving a WM_DELETE_WINDOW message should behave as if the user selected "delete window" from a hypothetical menu. They should perform any confirmation dialog with the user and, if they decide to complete the deletion, should do the following:

- Either change the window's state to Withdrawn (as described in Changing Window State) or destroy the window.
- Destroy any internal state associated with the window.

If the user aborts the deletion during the confirmation dialog, the client should ignore the message.

Clients are permitted to interact with the user and ask, for example, whether a file associated with the window to be deleted should be saved or the window deletion should be cancelled. Clients are not required to destroy the window itself; the resource may be reused, but all associated state (for example, backing store) should be released.

If the client aborts a destroy and the user then selects DELETE WINDOW again, the window manager should start the WM_DELETE_WINDOW protocol again. Window managers should not use `DestroyWindow` requests on a window that has WM_DELETE_WINDOW in its WM_PROTOCOLS property.

Clients that choose not to include WM_DELETE_WINDOW in the WM_PROTOCOLS property may be disconnected from the server if the user asks for one of the client's top-level windows to be deleted.

Redirecting Requests

Normal clients can use the redirection mechanism just as window managers do by selecting for `SubstructureRedirect` events on a parent window or `ResizeRedirect` events on a window itself. However, at most, one client per window can select for these events, and a convention is needed to avoid clashes.

Convention

Clients (including window managers) should select for `SubstructureRedirect` and `ResizeRedirect` events only on windows that they own.

In particular, clients that need to take some special action if they are resized can select for `ResizeRedirect` events on their top-level windows. They will receive a `ResizeRequest` event if the window manager resizes their window, and the resize will not actually take place. Clients are free to make what use they like of the information that the window manager wants to change their size, but they must configure the window to the width and height specified in the event in a timely fashion. To ensure that the resize will actually happen at this stage instead of being intercepted and executed by the window manager (and thus restarting the process), the client needs temporarily to set override-redirect on the window.

Convention

Clients receiving `ResizeRequest` events must respond by doing the following:

- Setting override-redirect on the window specified in the event
- Configuring the window specified in the event to the width and height specified in the event as soon as possible and before making any other geometry requests
- Clearing override-redirect on the window specified in the event

If a window manager detects that a client is not obeying this convention, it is free to take whatever measures it deems appropriate to deal with the client.

Communication with the Window Manager by Means of Selections

For each screen they manage, window managers will acquire ownership of a selection named WM_S*n*, where *n* is the screen number, as described in Discriminated Names Window managers should comply with the conventions for "Manager Selections" described in Manager Selections. The intent is for clients to be able to request a variety of information or services by issuing conversion requests on this selection. Window managers should support conversion of the following target on their manager selection:

Atom	**Type**	**Data Received**
VERSION	INTEGER	Two integers, which are the major and minor release numbers (respectively) of the ICCCM with which the window manager complies. For this version of the ICCCM, the numbers are 2 and 0. [a]

---

[a] As a special case, clients not wishing to implement a selection request may simply issue a `GetSelectionOwner` request on the appropriate WM_S*n* selection. If this selection is owned, clients may assume that the window manager complies with ICCCM version 2.0 or later.

Summary of Window Manager Property Types

The window manager properties are summarized in the following table (see also section 14.1 of *Xlib - C Language X Interface*).

Name	Type	Format	See Section
WM_CLASS	STRING	8	WM_CLASS Property
WM_CLIENT_MACHINE	TEXT		WM_CLIENT_MACHINE Property
WM_COLORMAP_WINDOWS	WINDOW	32	WM_COLORMAP_WINDOWS Property
WM_HINTS	WM_HINTS	32	WM_HINTS Property
WM_ICON_NAME	TEXT		WM_ICON_NAME Property
WM_ICON_SIZE	WM_ICON_SIZE	32	WM_ICON_SIZE Property
WM_NAME	TEXT		WM_NAME Property
WM_NORMAL_HINTS	WM_SIZE_HINTS	32	WM_NORMAL_HINTS Property
WM_PROTOCOLS	ATOM	32	WM_PROTOCOLS Property
WM_STATE	WM_STATE	32	WM_STATE Property
WM_TRANSIENT_FOR	WINDOW	32	WM_TRANSIENT_FOR Property

Chapter 5. Session Management and Additional Inter-Client Exchanges

This section contains some conventions for clients that participate in session management. See *X Session Management Protocol* for further details. Clients that do not support this protocol cannot expect their window state (e.g., WM_STATE, position, size, and stacking order) to be preserved across sessions.

Client Support for Session Management

Each session participant will obtain a unique client identifier (client-ID) from the session manager. The client must identify one top-level window as the "client leader." This window must be created by the client. It may be in any state, including the Withdrawn state. The client leader window must have a SM_CLIENT_ID property, which contains the client-ID obtained from the session management protocol. That property must:

- Be of type STRING
- Be of format 8
- Contain the client-ID as a string of XPCS characters encoded using ISO 8859-1

All top-level, nontransient windows created by a client on the same display as the client leader must have a WM_CLIENT_LEADER property. This property contains a window ID that identifies the client leader window. The client leader window must have a WM_CLIENT_LEADER property containing its own window ID (i.e., the client leader window is pointing to itself). Transient windows need not have a WM_CLIENT_LEADER property if the client leader can be determined using the information in the WM_TRANSIENT_FOR property. The WM_CLIENT_LEADER property must:

- Be of type WINDOW
- Be of format 32
- Contain the window ID of the client leader window

A client must withdraw all of its top-level windows on the same display before modifiying either the WM_CLIENT_LEADER or the SM_CLIENT_ID property of its client leader window.

It is necessary that other clients be able to uniquely identify a window (across sessions) among all windows related to the same client-ID. For example, a window manager can require this unique ID to restore geometry information from a previous session, or a workspace manager could use it to restore information about which windows are in which workspace. A client may optionally provide a WM_WINDOW_ROLE property to uniquely identify a window within the scope specified above. The combination of SM_CLIENT_ID and WM_WINDOW_ROLE can be used by other clients to uniquely identify a window across sessions.

If the WM_WINDOW_ROLE property is not specified on a top-level window, a client that needs to uniquely identify that window will try to use instead the values of WM_CLASS and WM_NAME. If a client has multiple windows with identical WM_CLASS and WM_NAME properties, then it should provide a WM_WINDOW_ROLE property.

The client must set the WM_WINDOW_ROLE property to a string that uniquely identifies that window among all windows that have the same client leader window. The property must:

- Be of type STRING
- Be of format 8
- Contain a string restricted to the XPCS characters, encoded in ISO 8859-1

Window Manager Support for Session Management

A window manager supporting session management must register with the session manager and obtain its own client-ID. The window manager should save and restore information such as the WM_STATE, the layout of windows on the screen, and their stacking order for every client window that has a valid SM_CLIENT_ID property (on itself, or on the window named by WM_CLIENT_LEADER) and that can be uniquely identified. Clients are allowed to change this state during the first phase of the session checkpoint process. Therefore, window managers should request a second checkpoint phase and save clients' state only during that phase.

Support for ICE Client Rendezvous

The Inter-Client Exchange protocol (ICE) defined as of X11R6 specifies a generic communication framework, independent of the X server, for data exchange between arbitrary clients. ICE also defines a protocol for any two ICE clients who also have X connections to the same X server to locate (rendezvous with) each other.

This protocol, called the "ICE X Rendezvous" protocol, is defined in the ICE specification, Appendix B, and uses the property ICE_PROTOCOLS plus `ClientMessage` events. Refer to that specification for complete details.

Chapter 6. Manipulation of Shared Resources

X Version 11 permits clients to manipulate a number of shared resources, for example, the input focus, the pointer, and colormaps. Conventions are required so that clients share resources in an orderly fashion.

The Input Focus

Clients that explicitly set the input focus must observe one of two modes:

- Locally active mode
- Globally active mode

Conventions

- Locally active clients should set the input focus to one of their windows only when it is already in one of their windows or when they receive a WM_TAKE_FOCUS message. They should set the input field of the WM_HINTS structure to `True`.
- Globally active clients should set the input focus to one of their windows only when they receive a button event and a passive-grabbed key event, or when they receive a WM_TAKE_FOCUS message. They should set the input field of the WM_HINTS structure to `False`.
- In addition, clients should use the timestamp of the event that caused them to attempt to set the input focus as the time field on the `SetInputFocus` request, not `CurrentTime`.

The Pointer

In general, clients should not warp the pointer. Window managers, however, may do so (for example, to maintain the invariant that the pointer is always in the window with the input focus). Other window managers may want to preserve the illusion that the user is in sole control of the pointer.

Conventions

- Clients should not warp the pointer.
- Clients that insist on warping the pointer should do so only with the src-window argument of the `WarpPointer` request set to one of their windows.

Grabs

A client's attempt to establish a button or a key grab on a window will fail if some other client has already established a conflicting grab on the same window. The grabs, therefore, are shared resources, and their use requires conventions.

In conformance with the principle that clients should behave, as far as possible, when a window manager is running as they would when it is not, a client that has the input focus may assume that it can receive all the available keys and buttons.

Convention

> Window managers should ensure that they provide some mechanism for their clients to receive events from all keys and all buttons, except for events involving keys whose KeySyms are registered as being for window management functions (for example, a hypothetical WINDOW KeySym).

In other words, window managers must provide some mechanism by which a client can receive events from every key and button (regardless of modifiers) unless and until the X Consortium registers some KeySyms as being reserved for window management functions. Currently, no KeySyms are registered for window management functions.

Even so, clients are advised to allow the key and button combinations used to elicit program actions to be modified, because some window managers may choose not to observe this convention or may not provide a convenient method for the user to transmit events from some keys.

Convention

> Clients should establish button and key grabs only on windows that they own.

In particular, this convention means that a window manager that wishes to establish a grab over the client's top-level window should either establish the grab on the root or reparent the window and establish the grab on a proper ancestor. In some cases, a window manager may want to consume the event received, placing the window in a state where a subsequent such event will go to the client. Examples are:

- Clicking in a window to set focus with the click not being offered to the client
- Clicking in a buried window to raise it, again, with the click not offered to the client

More typically, a window manager should add to, rather than replace, the client's semantics for key+button combinations by allowing the event to be used by the client after the window manager is done with it. To ensure this, the window manager should establish the grab on the parent by using the following:

```
pointer/keyboard-mode == Synchronous
```

Then, the window manager should release the grab by using an `AllowEvents` request with the following specified:

```
mode == ReplayPointer/Keyboard
```

In this way, the client will receive the events as if they had not been intercepted.

Obviously, these conventions place some constraints on possible user interface policies. There is a trade-off here between freedom for window managers to implement their user interface policies and freedom for clients to implement theirs. The dilemma is resolved by:

- Allowing window managers to decide if and when a client will receive an event from any given key or button
- Placing a requirement on the window manager to provide some mechanism, perhaps a "Quote" key, by which the user can send an event from any key or button to the client

Colormaps

Colormaps prescribes conventions for clients to communicate with the window manager about their colormap needs. If your clients are `DirectColor` type applications, you should consult section 14.3 of *Xlib - C Language X Interface* for conventions connected with sharing standard colormaps. They should look for and create the properties described there on the root window of the appropriate screen.

The contents of the RGB_COLOR_MAP type property are as follows:

Field	**Type**	**Comments**
colormap	COLORMAP	ID of the colormap described
red_max	CARD32	Values for pixel calculations
red_mult	CARD32	
green_max	CARD32	
green_mult	CARD32	
blue_max	CARD32	
blue_mult	CARD32	
base_pixel	CARD32	
visual_id	VISUALID	Visual to which colormap belongs
kill_id	CARD32	ID for destroying the resources

When deleting or replacing an RGB_COLOR_MAP, it is not sufficient to delete the property; it is important to free the associated colormap resources as well. If kill_id is greater than one, the resources should be freed by issuing a `KillClient` request with kill_id as the argument. If kill_id is one, the resources should be freed by issuing a `FreeColormap` request with colormap as the colormap argument. If kill_id is zero, no attempt should be made to free the resources. A client that creates an RGB_COLOR_MAP for which the colormap resource is created specifically for this purpose should set kill_id to one (and can create more than one such standard colormap using a single connection). A client that creates an RGB_COLOR_MAP for which the colormap resource is shared in some way (for example, is the default colormap for the root window) should create an arbitrary resource and use its resource ID for kill_id (and should create no other standard colormaps on the connection).

Convention

If an RGB_COLOR_MAP property is too short to contain the visual_id field, it can be assumed that the visual_id is the root visual of the appropriate screen. If an RGB_COLOR_MAP property is too short to contain the kill_id field, a value of zero can be assumed.

During the connection handshake, the server informs the client of the default colormap for each screen. This is a colormap for the root visual, and clients can use it to improve the extent of colormap sharing if they use the root visual.

The Keyboard Mapping

The X server contains a table (which is read by `GetKeyboardMapping` requests) that describes the set of symbols appearing on the corresponding key for each keycode generated by the server. This table does not affect the server's operations in any way; it is simply a database used by clients that attempt to understand the keycodes they receive. Nevertheless, it is a shared resource and requires conventions.

It is possible for clients to modify this table by using a `ChangeKeyboardMapping` request. In general, clients should not do this. In particular, this is not the way in which clients should implement key bindings or key remapping. The conversion between a sequence of keycodes received from the server and a string in a particular encoding is a private matter for each client (as it must be in a world where applications may be using different encodings to support different languages and fonts). See the Xlib reference manual for converting keyboard events to text.

The only valid reason for using a `ChangeKeyboardMapping` request is when the symbols written on the keys have changed as, for example, when a Dvorak key conversion kit or a set of APL keycaps has been installed. Of course, a client may have to take the change to the keycap on trust.

The following illustrates a permissible interaction between a client and a user:

- "You just started me on a server without a Pause key. Please choose a key to be the Pause key and press it now."
- Presses the Scroll Lock key
- "Adding Pause to the symbols on the Scroll Lock key: Confirm or Abort."
- Confirms
- Uses a `ChangeKeyboardMapping` request to add Pause to the keycode that already contains Scroll Lock and issues this request, "Please paint Pause on the Scroll Lock key." Clients should not use `ChangeKeyboardMapping` requests.

If a client succeeds in changing the keyboard mapping table, all clients will receive `MappingNotify` (request==Keyboard) events. There is no mechanism to avoid receiving these events.

Convention

Clients receiving `MappingNotify` (request==Keyboard) events should update any internal keycode translation tables they are using.

The Modifier Mapping

X Version 11 supports 8 modifier bits of which 3 are preassigned to Shift, Lock, and Control. Each modifier bit is controlled by the state of a set of keys, and these sets are specified in a table accessed by `GetModifierMapping` and `SetModifierMapping` requests. This table is a shared resource and requires conventions.

A client that needs to use one of the preassigned modifiers should assume that the modifier table has been set up correctly to control these modifiers. The Lock modifier should be interpreted as Caps Lock or Shift Lock according as the keycodes in its controlling set include XK_Caps_Lock or XK_Shift_Lock.

Convention

> Clients should determine the meaning of a modifier bit from the KeySyms being used to control it.

A client that needs to use an extra modifier (for example, META) should do the following:

- Scan the existing modifier mappings. If it finds a modifier that contains a keycode whose set of KeySyms includes XK_Meta_L or XK_Meta_R, it should use that modifier bit.
- If there is no existing modifier controlled by XK_Meta_L or XK_Meta_R, it should select an unused modifier bit (one with an empty controlling set) and do the following:
  - If there is a keycode with XL_Meta_L in its set of KeySyms, add that keycode to the set for the chosen modifier.
  - If there is a keycode with XL_Meta_R in its set of KeySyms, add that keycode to the set for the chosen modifier.
  - If the controlling set is still empty, interact with the user to select one or more keys to be META.
- If there are no unused modifier bits, ask the user to take corrective action.

Conventions

> - Clients needing a modifier not currently in use should assign keycodes carrying suitable KeySyms to an unused modifier bit.
> - Clients assigning their own modifier bits should ask the user politely to remove his or her hands from the key in question if their `SetModifierMapping` request returns a `Busy` status.

There is no good solution to the problem of reclaiming assignments to the five non-preassigned modifiers when they are no longer being used.

Convention

> The user must use `xmodmap` or some other utility to deassign obsolete modifier mappings by hand.

When a client succeeds in performing a `SetModifierMapping` request, all clients will receive `MappingNotify` (request==Modifier) events. There is no mechanism for preventing these events from being received. A client that uses one of the nonpreassigned modifiers that receives one of these events should do a `GetModifierMap-`

`ping` request to discover the new mapping, and if the modifier it is using has been cleared, it should reinstall the modifier.

Note that a `GrabServer` request must be used to make the `GetModifierMapping` and `SetModifierMapping` pair in these transactions atomic.

Chapter 7. Device Color Characterization

The X protocol provides explicit Red, Green, and Blue (RGB) values, which are used to directly drive a monitor, and color names. RGB values provide a mechanism for accessing the full capabilities of the display device, but at the expense of having the color perceived by the user remain unknowable through the protocol. Color names were originally designed to provide access to a device-independent color database by having the server vendor tune the definitions of the colors in that textual database. Unfortunately, this still does not provide the client any way of using an existing device-independent color, nor for the client to get device-independent color information back about colors that it has selected.

Furthermore, the client must be able to discover which set of colors are displayable by the device (the device gamut), both to allow colors to be intelligently modified to fit within the device capabilities (gamut compression) and to enable the user interface to display a representation of the reachable color space to the user (gamut display).

Therefore, a system is needed that will provide full access to device-independent color spaces for X clients. This system should use a standard mechanism for naming the colors, be able to provide names for existing colors, and provide means by which unreachable colors can be modified to fall within the device gamut.

We are fortunate in this area to have a seminal work, the 1931 CIE color standard, which is nearly universally agreed upon as adequate for describing colors on CRT devices. This standard uses a tri-stimulus model called CIE XYZ in which each perceivable color is specified as a triplet of numbers. Other appropriate device-independent color models do exist, but most of them are directly traceable back to this original work.

X device color characterization provides device-independent color spaces to X clients. It does this by providing the barest possible amount of information to the client that allows the client to construct a mapping between CIE XYZ and the regular X RGB color descriptions.

Device color characterization is defined by the name and contents of two window properties that, together, permit converting between CIE XYZ space and linear RGB device space (such as standard CRTs). Linear RGB devices require just two pieces of information to completely characterize them:

- A 3 x 3 matrix M and its inverse M^{-1}, which convert between XYZ and RGB intensity ($RGB_{intensity}$):

  $RGB_{intensity} = M \times XYZ$

  $XYZ = M^{-1} \times RGB_{intensity}$

- A way of mapping between RGB intensity and RGB protocol value. XDCCC supports three mechanisms which will be outlined later.

If other device types are eventually necessary, additional properties will be required to describe them.

XYZ <-> RGB Conversion Matrices

Because of the limited dynamic range of both XYZ and RGB intensity, these matrices will be encoded using a fixed-point representation of a 32-bit two's complement number scaled by 2^{27}, giving a range of -16 to 16 - ε, where $ε = 2^{-27}$.

These matrices will be packed into an 18-element list of 32-bit values, XYZ -> RGB matrix first, in row major order and stored in the XDCCC_LINEAR_RGB_MATRICES properties (format = 32) on the root window of each screen, using values appropriate for that screen.

This will be encoded as shown in the following table:

Field	Type	Comments
$M_{0,0}$	INT32	Interpreted as a fixed-point number -16 ≤ x < 16
$M_{0,1}$	INT32	
...	INT32	
$M_{3,3}$	INT32	
$M^{-1}{}_{0,0}$	INT32	
$M^{-1}{}_{0,1}$	INT32	
...	INT32	
$M^{-1}{}_{3,3}$	INT32	

Intensity (dA RGB Value Conversion

XDCCC provides two representations for describing the conversion between RGB intensity and the actual X protocol RGB values:

```
0     RGB value/RGB intensity level pairs
1     RGB intensity ramp
```

In both cases, the relevant data will be stored in the XDCCC_LINEAR_RGB_CORRECTION properties on the root window of each screen, using values appropriate for that screen, in whatever format provides adequate resolution. Each property can consist of multiple entries concatenated together, if different visuals for the screen require different conversion data. An entry with a VisualID of 0 specifies data for all visuals of the screen that are not otherwise explicitly listed.

The first representation is an array of RGB value/intensity level pairs, with the RGB values in strictly increasing order. When converting, the client must linearly interpolate between adjacent entries in the table to compute the desired value. This allows the server to perform gamma correction itself and encode that fact in a short two-element correction table. The intensity will be encoded as an unsigned number to be interpreted as a value between 0 and 1 (inclusive). The precision of this value will depend on the format of the property in which it is stored (8, 16, or 32 bits). For 16-bit and 32-bit formats, the RGB value will simply be the value stored in the property. When stored in 8-bit format, the RGB value can be computed from the value in the property by:

RGB sub value ~ = ~ { Property ~ Value ~ times ~ 65535 } over 255

Because the three electron guns in the device may not be exactly alike in response characteristics, it is necessary to allow for three separate tables, one each for red, green, and blue. Therefore, each table will be preceded by the number of entries in that table, and the set of tables will be preceded by the number of tables. When three tables are provided, they will be in red, green, blue order.

This will be encoded as shown in the following table:

XDCCC_LINEAR_RGB_CORRECTION Property Contents for Type 0 Correction

Field	**Type**	**Comments**
VisualID0	CARD	Most significant portion of VisualID
VisualID1	CARD	Exists if and only if the property format is 8
VisualID2	CARD	Exists if and only if the property format is 8
VisualID3	CARD	Least significant portion, exists if and only if the property format is 8 or 16
type	CARD	0 for this type of correction
count	CARD	Number of tables following (either 1 or 3)
length	CARD	Number of pairs -1 following in this table
value	CARD	X Protocol RBG value
intensity	CARD	Interpret as number 0 ≤ *intensity* ≤ 1
...	...	Total of *length+1* pairs of value/intensity values
lengthg	CARD	Number of pairs -1 following in this table (if and only if *count* is 3
value	CARD	X Protocol RBG value
intensity	CARD	Interpret as a number 0 ≤ *intensity* ≤ 1
...	...	Total of *length+1* pairs of value/intensity values
lengthb	CARD	Number of pairs -1 following in this table (if and only if *count* is 3
value	CARD	X Protocol RBG value
intensity	CARD	Interpret as a number 0 ≤ *intensity* ≤ 1
...	...	Total of *length+1* pairs of value/intensity values

The VisualID is stored in 4, 2, or 1 pieces, depending on whether the property format is 8, 16, or 32, respectively. The VisualID is always stored most significant piece first. Note that the length fields are stored as one less than the actual length, so 256 entries can be stored in format 8.

The second representation is a simple array of intensities for a linear subset of RGB values. The expected size of this table is the bits-per-rgb-value of the screen, but it can be any length. This is similar to the first mechanism, except that the RGB value numbers are implicitly defined by the index in the array (indices start at 0):

> RGB sub value ~ = ~ { Array ~ Index ~ times ~ 65535 } over { Array ~ Size ~ - ~ 1 }

When converting, the client may linearly interpolate between entries in this table. The intensity values will be encoded just as in the first representation.

This will be encoded as shown in the following table:

XDCCC_LINEAR_RGB_CORRECTION Property Contents for Type 1 Correction

Field	**Type**	**Comments**
VisualID0	CARD	Most significant portion of VisualID
VisualID1	CARD	Exists if and only if the property format is 8
VisualID2	CARD	Exists if and only if the property format is 8
VisualID3	CARD	Least significant portion, exists if and only if the property format is 8 or 16
type	CARD	1 for this type of correction
count	CARD	Number of tables following (either 1 or 3)
length	CARD	Number of pairs -1 following in this table
intensity	CARD	Interpret as number 0 ≤ *intensity* ≤ 1
...	...	Total of *length+1* pairs of value/intensity values
lengthg	CARD	Number of pairs -1 following in this table (if and only if *count* is 3
intensity	CARD	Interpret as a number 0 ≤ *intensity* ≤ 1
...	...	Total of *length+1* pairs of value/intensity values
lengthb	CARD	Number of pairs -1 following in this table (if and only if *count* is 3
intensity	CARD	Interpret as a number 0 ≤ *intensity* ≤ 1
...	...	Total of *length+1* pairs of value/intensity values

Chapter 8. Conclusion

This document provides the protocol-level specification of the minimal conventions needed to ensure that X Version 11 clients can interoperate properly. This document specifies interoperability conventions only for the X Version 11 protocol. Clients should be aware of other protocols that should be used for better interoperation in the X environment. The reader is referred to *X Session Management Protocol* for information on session management, and to *Inter-Client Exchange Protocol* for information on general-purpose communication among clients.

The X Registry

The X Consortium maintains a registry of certain X-related items, to aid in avoiding conflicts and in sharing of such items. Readers are encouraged to use the registry. The classes of items kept in the registry that are relevant to the ICCCM include property names, property types, selection names, selection targets, WM_PROTOCOLS protocols, `ClientMessage` types, and application classes. Requests to register items, or questions about registration, should be addressed to

xregistry@x.org

or to

The X.Org Foundation -- X11 Registry
c/o Alan Coopersmith
Oracle Corporation
M/S SCA17-3824
4170 Network Circle
Santa Clara, CA 95054
USA

Electronic mail will be acknowledged upon receipt. Please allow up to 4 weeks for a formal response to registration and inquiries.

The registry is published as part of the X software distribution from the X.Org Foundation. All registered items must have the postal address of someone responsible for the item or a reference to a document describing the item and the postal address of where to write to obtain the document.

Appendix A. Revision History

This appendix describes the revision history of this document and summarizes the incompatibilities between this and earlier versions.

The X11R2 Draft

The February 25, 1988, draft that was distributed as part of X Version 11, Release 2, was clearly labeled as such, and many areas were explicitly labeled as liable to change. Nevertheless, in the revision work done since then, we have been very careful not to introduce gratuitous incompatibility. As far as possible, we have tried to ensure that clients obeying the conventions in the X11R2 draft would still work.

The July 27, 1988, Draft

The Consortium review was based on a draft dated July 27, 1988. This draft included several areas in which incompatibilities with the X11R2 draft were necessary:

- The use of property `None` in `ConvertSelection` requests is no longer allowed. Owners that receive them are free to use the target atom as the property to respond with, which will work in most cases.
- The protocol for INCREMENTAL type properties as selection replies has changed, and the name has been changed to INCR. Selection requestors are free to implement the earlier protocol if they receive properties of type INCREMENTAL.
- The protocol for INDIRECT type properties as selection replies has changed, and the name has been changed to MULTIPLE. Selection requestors are free to implement the earlier protocol if they receive properties of type INDIRECT.
- The protocol for the special CLIPBOARD client has changed. The earlier protocol is subject to race conditions and should not be used.
- The set of state values in WM_HINTS.initial_state has been reduced, but the values that are still valid are unchanged. Window managers should treat the other values sensibly.
- The methods an application uses to change the state of its top-level window have changed but in such a way that cases that used to work will still work.
- The x, y, width, and height fields have been removed from the WM_NORMAL_HINTS property and replaced by pad fields. Values set into these fields will be ignored. The position and size of the window should be set by setting the appropriate window attributes.
- A pair of base fields and a win_gravity field have been added to the WM_NORMAL_HINTS property. Window managers will assume values for these fields if the client sets a short property.

The Public Review Drafts

The Consortium review resulted in several incompatible changes. These changes were included in drafts that were distributed for public review during the first half of 1989.

- The messages field of the WM_HINTS property was found to be unwieldy and difficult to evolve. It has been replaced by the WM_PROTOCOLS property, but clients that use the earlier mechanism can be detected because they set the messages bit in the flags field of the WM_HINTS property, and window managers can provide a backwards compatibility mode.

- The mechanism described in the earlier draft by which clients installed their own subwindow colormaps could not be made to work reliably and mandated some features of the look and feel. It has been replaced by the WM_COLORMAP_WINDOWS property. Clients that use the earlier mechanism can be detected by the WM_COLORMAPS property they set on their top-level window, but providing a reliable backwards compatibility mode is not possible.

- The recommendations for window manager treatment of top-level window borders have been changed as those in the earlier draft produced problems with Visibility events. For nonwindow manager clients, there is no incompatibility.

- The pseudoroot facility in the earlier draft has been removed. Although it has been successfully implemented, it turns out to be inadequate to support the uses envisaged. An extension will be required to support these uses fully, and it was felt that the maximum freedom should be left to the designers of the extension. In general, the previous mechanism was invisible to clients and no incompatibility should result.

- The addition of the WM_DELETE_WINDOW protocol (which prevents the danger that multi-window clients may be terminated unexpectedly) has meant some changes in the WM_SAVE_YOURSELF protocol, to ensure that the two protocols are orthogonal. Clients using the earlier protocol can be detected (see WM_PROTOCOLS above) and supported in a backwards compatibility mode.

- The conventions in Section 14.3.1. of *Xlib - C Language X Interface* regarding properties of type RGB_COLOR_MAP have been changed, but clients that use the earlier conventions can be detected because their properties are 4 bytes shorter. These clients will work correctly if the server supports only a single Visual or if they use only the Visual of the root. These are the only cases in which they would have worked, anyway.

Version 1.0, July 1989

The public review resulted in a set of mostly editorial changes. The changes in version 1.0 that introduced some degree of incompatibility with the earlier drafts are:

- A new section (Grabs) was added covering the window manager's use of Grabs. The restrictions it imposes should affect only window managers.

- The TARGETS selection target has been clarified, and it may be necessary for clients to add some entries to their replies.

- A selection owner using INCR transfer should no longer replace targets in a MULTIPLE property with the atom INCR.

- The contents of the `ClientMessage` event sent by a client to iconify itself has been clarified, but there should be no incompatibility because the earlier contents would not in fact have worked.

- The border-width in synthetic `ConfigureNotify` events is now specified, but this should not cause any incompatibility.
- Clients are now asked to set a border-width on all `ConfigureWindow` requests.
- Window manager properties on icon windows now will be ignored, but there should be no incompatibility because there was no specification that they be obeyed previously.
- The ordering of real and synthetic `ConfigureNotify` events is now specified, but any incompatibility should affect only window managers.
- The semantics of WM_SAVE_YOURSELF have been clarified and restricted to be a checkpoint operation only. Clients that were using it as part of a shutdown sequence may need to be modified, especially if they were interacting with the user during the shutdown.
- A kill_id field has been added to RGB_COLOR_MAP properties. Clients using earlier conventions can be detected by the size of their RGB_COLOR_MAP properties, and the cases that would have worked will still work.

Version 1.1

Version 1.1 was released with X11R5 in September 1991. In addition to some minor editorial changes, there were a few semantic changes since Version 1.0:

- The section on Device Color Characterization was added.
- The meaning of the NULL property type was clarified.
- Appropriate references to Compound Text were added.

Public Review Draft, December 1993

The following changes have been made in preparing the public review draft for Version 2.0.

- [P01] Addition of advice to clients on how to keep track of a top-level window's absolute position on the screen.
- [P03] A technique for clients to detect when it is safe to reuse a top-level window has been added.
- [P06] Colormaps , on colormaps, has been rewritten. A new feature that allows clients to install their own colormaps has also been added.
- [P08] The LENGTH target has been deprecated.
- [P11] The manager selections facility was added.
- [P17] The definition of the aspect ratio fields of the WM_NORMAL_HINTS property has been changed to include the base size.
- [P19] `StaticGravity` has been added to the list of values allowed for the win_gravity field of the WM_HINTS property. The meaning of the `CenterGravity` value has been clarified.

- [P20] A means for clients to query the ICCCM compliance level of the window manager has been added.
- [P22] The definition of the MULTIPLE selection target has been clarified.
- [P25] A definition of "top-level window" has been added. The WM_STATE property has been defined and exposed to clients.
- [P26] The definition of window states has been clarified and the wording regarding window state changes has been made more consistent.
- [P27] Clarified the rules governing when window managers are required to send synthetic `ConfigureNotify` events.
- [P28] Added a recommended technique for setting the input focus to a window as soon as it is mapped.
- [P29] The required lifetime of resource IDs named in window manager properties has been specified.
- [P30] Advice for dealing with keystrokes and override-redirect windows has been added.
- [P31] A statement on the ownership of resources transferred through the selection mechanism has been added.
- [P32] The definition of the CLIENT_WINDOW target has been clarified.
- [P33] A rule about requiring the selection owner to reacquire the selection under certain circumstances has been added.
- [P42] Added several new selection targets.
- [P44] Ambiguous wording regarding the withdrawal of top-level windows has been removed.
- [P45] A facility for requestors to pass parameters during a selection request has been added.
- [P49] A convention on discrimated names has been added.
- [P57] The C_STRING property type was added.
- [P62] An ordering requirement on processing selection requests was added.
- [P63] The `VisibleHint` flag was added.
- [P64] The session management section has been updated to align with the new session management protocol. The old session management conventions have been moved to Appendix C.
- References to the never-forthcoming *Window and Session Manager Conventions Manual* have been removed.
- Information on the X Registry and references to the session management and ICE documents have been added.
- Numerous editorial and typographical improvements have been made.

Version 2.0, April 1994

The following changes have been made in preparation for releasing the final edition of Version 2.0 with X11R6.

- The PIXMAP selection target has been revised to return a property of type PIXMAP instead of type DRAWABLE.
- The session management section has been revised slightly to correspond with the changes to the *X Session Management Protocol*.
- Window managers are now prohibited from placing `CurrentTime` in the timestamp field of WM_TAKE_FOCUS messages.
- In the WM_HINTS property, the `VisibleHint` flag has been renamed to `UrgencyHint`. Its semantics have also been defined more thoroughly.
- Additional editorial and typographical changes have been made.

Appendix B. Suggested Protocol Revisions

During the development of these conventions, a number of inadequacies have been discovered in the core X11 protocol. They are summarized here as input to an eventual protocol revision design process:

- There is no way for anyone to find out the last-change time of a selection. The `GetSelectionOwner` request should be changed to return the last-change time as well as the owner.

- There is no way for a client to find out which selection atoms are valid.

- There would be no need for WM_TAKE_FOCUS if the `FocusIn` event contained a timestamp and a previous-focus field. This could avoid the potential race condition. There is space in the event for this information; it should be added at the next protocol revision.

- There is a race condition in the `InstallColormap` request. It does not take a timestamp and may be executed after the top-level colormap has been uninstalled. The next protocol revision should provide the timestamp in the `InstallColormap`, `UninstallColormap`, `ListInstalledColormaps` requests and in the `ColormapNotify` event. The timestamp should be used in a similar way to the last-focus-change time for the input focus. The lack of timestamps in these packets is the reason for restricting colormap installation to the window manager.

- The protocol needs to be changed to provide some way of identifying the Visual and the Screen of a colormap.

- There should be some way to reclaim assignments to the five nonpreassigned modifiers when they are no longer needed. The manual method is unpleasantly low-tech.

Appendix C. Obsolete Session Manager Conventions

This appendix contains obsolete conventions for session management using X properties and messages. The conventions described here are deprecated and are described only for historical interest. For further information on session management, see *X Session Management Protocol*.

Properties

The client communicates with the session manager by placing two properties (WM_COMMAND and WM_CLIENT_MACHINE) on its top-level window. If the client has a group of top-level windows, these properties should be placed on the group leader window.

The window manager is responsible for placing a WM_STATE property on each top-level client window for use by session managers and other clients that need to be able to identify top-level client windows and their state.

WM_COMMAND Property

The WM_COMMAND property represents the command used to start or restart the client. By updating this property, clients should ensure that it always reflects a command that will restart them in their current state. The content and type of the property depend on the operating system of the machine running the client. On POSIX-conformant systems using ISO Latin-1 characters for their command lines, the property should:

- Be of type STRING
- Contain a list of null-terminated strings
- Be initialized from argv

  Other systems will need to set appropriate conventions for the type and contents of WM_COMMAND properties. Window and session managers should not assume that STRING is the type of WM_COMMAND or that they will be able to understand or display its contents.

Note that WM_COMMAND strings are null-terminated and differ from the general conventions that STRING properties are null-separated. This inconsistency is necessary for backwards compatibility.

A client with multiple top-level windows should ensure that exactly one of them has a WM_COMMAND with nonzero length. Zero-length WM_COMMAND properties can be used to reply to WM_SAVE_YOURSELF messages on other top-level windows but will otherwise be ignored.

WM_CLIENT_MACHINE Property

This property is described in WM_CLIENT_MACHINE Property.

Termination

Because they communicate by means of unreliable network connections, clients must be prepared for their connection to the server to be terminated at any time without warning. They cannot depend on getting notification that termination is imminent or on being able to use the server to negotiate with the user about their fate. For example, clients cannot depend on being able to put up a dialog box.

Similarly, clients may terminate at any time without notice to the session manager. When a client terminates itself rather than being terminated by the session manager, it is viewed as having resigned from the session in question, and it will not be revived if the session is revived.

Client Responses to Session Manager Actions

Clients may need to respond to session manager actions in two ways:

- Saving their internal state
- Deleting a window

Saving Client State

Clients that want to be warned when the session manager feels that they should save their internal state (for example, when termination impends) should include the atom WM_SAVE_YOURSELF in the WM_PROTOCOLS property on their top-level windows to participate in the WM_SAVE_YOURSELF protocol. They will receive a `ClientMessage` event as described in ClientMessage Events with the atom WM_SAVE_YOURSELF in its data[0] field.

Clients that receive WM_SAVE_YOURSELF should place themselves in a state from which they can be restarted and should update WM_COMMAND to be a command that will restart them in this state. The session manager will be waiting for a `PropertyNotify` event on WM_COMMAND as a confirmation that the client has saved its state. Therefore, WM_COMMAND should be updated (perhaps with a zero-length append) even if its contents are correct. No interactions with the user are permitted during this process.

Once it has received this confirmation, the session manager will feel free to terminate the client if that is what the user asked for. Otherwise, if the user asked for the session to be put to sleep, the session manager will ensure that the client does not receive any mouse or keyboard events.

After receiving a WM_SAVE_YOURSELF, saving its state, and updating WM_COMMAND, the client should not change its state (in the sense of doing anything that would require a change to WM_COMMAND) until it receives a mouse or keyboard event. Once it does so, it can assume that the danger is over. The session manager will ensure that these events do not reach clients until the danger is over or until the clients have been killed.

Irrespective of how they are arranged in window groups, clients with multiple top-level windows should ensure the following:

- Only one of their top-level windows has a nonzero-length WM_COMMAND property.
- They respond to a WM_SAVE_YOURSELF message by:
  - First, updating the nonzero-length WM_COMMAND property, if necessary
  - Second, updating the WM_COMMAND property on the window for which they received the WM_SAVE_YOURSELF message if it was not updated in the first step

Receiving WM_SAVE_YOURSELF on a window is, conceptually, a command to save the entire client state. [1]

Window Deletion

Windows are deleted using the WM_DELETE_WINDOW protocol, which is described in Window Deletion.

Summary of Session Manager Property Types

The session manager properties are listed in the following table:

Name	**Type**	**Format**	**See Section**
WM_CLIENT_MACHINE	TEXT		WM_CLIENT_MACHINE Property
WM_COMMAND	TEXT		WM_COMMAND Property
WM_STATE	WM_STATE	32	WM_STATE Property

---

[1] This convention has changed since earlier drafts because of the introduction of the protocol in the next section. In the public review draft, there was ambiguity as to whether WM_SAVE_YOURSELF was a checkpoint or a shutdown facility. It is now unambiguously a checkpoint facility; if a shutdown facility is judged to be necessary, a separate WM_PROTOCOLS protocol will be developed and registered with the X Consortium.