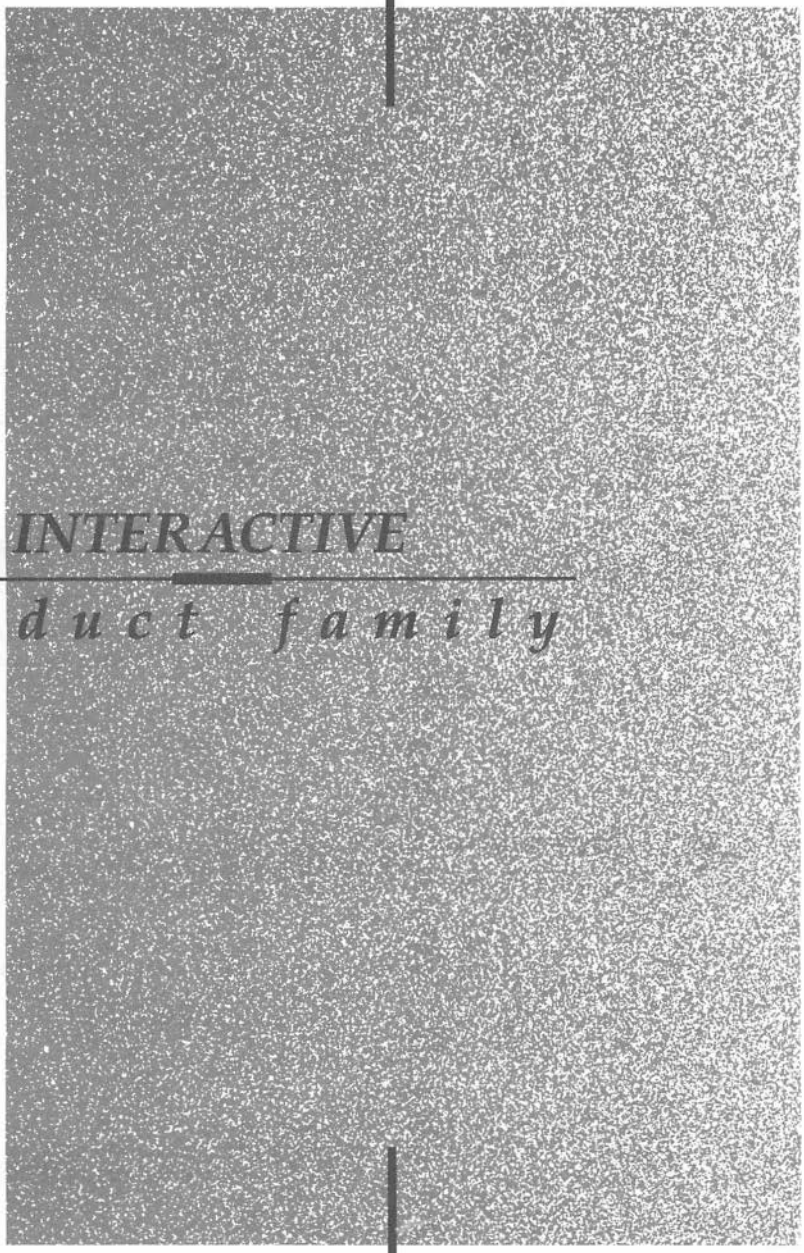
INTERACTIVE X11
Development System Guide



INTERACTIVE

First printing (October 1991)

INTERACTIVE Systems Corporation
2401 Colorado Avenue
Santa Monica, California 90404

INTERACTIVE X11
Development System Guide

CONTENTS

Inter-Client Communication Conventions Manual
INTERACTIVE TCP/IP Programmer's Supplement

Inter-Client Communication Conventions Manual
Version 1.0
MIT X Consortium Standard

CONTENTS

1. INTRODUCTION 2
   1.1 Evolution of the Conventions 2
   1.2 Atoms 3
      1.2.1 What Are Atoms? 3
      1.2.2 Predefined Atoms 4
      1.2.3 Naming Conventions 4
      1.2.4 Semantics 4
      1.2.5 Name Spaces 5

2. PEER-TO-PEER COMMUNICATION VIA SELECTIONS 6
   2.1 Acquiring Selection Ownership 7
   2.2 Responsibilities of the Selection Owner 8
   2.3 Giving Up Selection Ownership 10
      2.3.1 Voluntarily 10
      2.3.2 Forcibly 11
   2.4 Requesting a Selection 11
   2.5 Large Data Transfers 13
   2.6 Usage of Selection Atoms 14
      2.6.1 Selection Atoms 14
      2.6.2 Target Atoms 16
      2.6.3 Selection With Side Effects 19
   2.7 Usage of Selection Properties 20
      2.7.1 TEXT Properties 21
      2.7.2 INCR Properties 22
      2.7.3 DRAWABLE Properties 23
      2.7.4 SPAN Properties 23

3. PEER-TO-PEER COMMUNICATION VIA CUT-BUFFERS 24

4. CLIENT TO WINDOW MANAGER COMMUNICATION 25
   4.1 Client's Actions 26
      4.1.1 Creating a Top-Level Window 26

4.1.2 Client Properties 27
4.1.3 Window Manager Properties 36
4.1.4 Changing Window State 37
4.1.5 Configuring the Window 40
4.1.6 Changing Window Attributes 42
4.1.7 Input Focus 43
4.1.8 Colormaps 47
4.1.9 Icons 49
4.1.10 Pop-up Windows 51
4.1.11 Window Groups 51
4.2 Client Responses to Window Manager Actions 52
4.2.1 Reparenting 52
4.2.2 Redirection of Operations 53
4.2.3 Window Move 55
4.2.4 Window Resize 55
4.2.5 (De)Iconify 55
4.2.6 Colormap Change 56
4.2.7 Input Focus 56
4.2.8 ClientMessage Events 57
4.2.9 Redirecting Requests 58
4.3 Summary of Window Manager Property Types 58

5. CLIENT TO SESSION MANAGER COMMUNICATION 60
5.1 Client Actions 60
5.1.1 Properties 60
5.1.2 Termination 63
5.2 Client Responses to Session Manager Actions 63
5.2.1 Saving Client State 63
5.2.2 Window Deletion 64
5.3 Summary of Session Manager Property Types . . . 65

6. MANIPULATION OF SHARED RESOURCES . . . 67
6.1 The Input Focus 67
6.2 The Pointer 67
6.3 Grabs 67
6.4 Colormaps 69
6.5 The Keyboard Mapping 71
6.6 The Modifier Mapping 72

7. RESOURCE MANAGER CONVENTIONS 74

8. CONCLUSION 75
9. ACKNOWLEDGMENTS 76
Appendix A: COMPATIBILITY WITH EARLIER DRAFTS 77
1. A1: The R2 Draft 77
2. A2: The 27th July 1988 Draft 78
3. A3: The Public Review Drafts 79
Appendix B: SUGGESTED PROTOCOL REVISIONS 81

Inter-Client Communication Conventions Manual
Version 1.0
MIT X Consortium Standard

David S. H. Rosenthal
Sun Microsystems, Inc.

ABSTRACT

It was an explicit design goal of the X Window System, 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. Conventions are proposed to allow clients to cooperate in the areas of selections, cut-buffers, window management, session management, and resources.*

> ""Gentlemen." said Lord Chancellor Thurlow to the deputation of Nonconformists which waited on him in 1788 to ask for a repeal of the Corporation and Test Acts, "I'm against you, by God! I am for the Established Church, damme! Not that I have any more regard for the Established Church than for any other religion, but because it is established. And if you can get your damned religion established, I'll be for that too!""
>
> —*Years of Endurance*, Arthur Bryant

This document is reprinted (with editorial changes) with the permission of MIT and Sun Microsystems, Inc.

1. INTRODUCTION

It was an explicit design goal of X11 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 X11 world involves adhering to conventions governing inter-client communications in a number of areas:

- The selection mechanism
- The cut-buffers
- The window manager
- The session manager
- The manipulation of shared resources
- The resource database

In the following sections we propose suitable conventions for each area, in so far as it is possible to do so *without* enforcing a particular user interface. In order to permit clients written in different languages to communicate, the conventions are expressed solely in terms of the protocol operations, not in the (probably more familiar) Xlib interface. 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.

1.1 Evolution of the Conventions

In the interests of timely acceptance, this first edition of the manual covers only a minimal set of required conventions. It is expected that as experience is gained, these conventions will be added to, and conventions governing other, optional areas will be agreed upon. The X Consortium is expected to develop mechanisms for doing this.

As far as possible, these conventions are upwards-compatible with those in the 25th February 1988 draft of this manual distributed with the X11R2 release. In some areas, semantic problems were discovered with those conventions and thus complete upwards compatibility could not be assured. Areas of incompatibility are noted in the text and summarized in Appendix A.

In the course of developing these conventions, a number of minor changes to the protocol have been identified as desirable. They are identified in the text, and summarized in Appendix B, as input to a future protocol revision process. If and when a protocol revision incorporating them is undertaken, this document will need some revision. Since 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 the conventions will be revised in such a way as to ensure that clients using the conventions appropriate to protocol minor revision *n* will inter-operate correctly with those using conventions appropriate to protocol minor revision *n*+1 if the server supports both.

1.2 Atoms

Many of the conventions described below use Atoms. The following sections amplify the description of Atoms in the protocol specification, to assist the reader.

1.2.1 What Are Atoms?

At the conceptual level, Atoms are unique names. Clients can use them 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, it is important that events as they appear "on the wire" have a fixed size (in fact, 32 bytes), and since some events contain Atoms, a fixed-size representation for them was needed.

To provide a fixed-size representation, a protocol request (**InternAtom**) was provided to register a byte sequence with the server,

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.

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**).

1.2.2 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 will eliminate many **InternAtom** requests in most applications. Note that "predefined" is only in the sense of having numeric values, not in the sense of having required semantics."

They are an implementation trick to avoid the cost of Interning many 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/atom cache and will typically prefill this cache with the predefined atoms.

1.2.3 Naming Conventions

The built-in atoms are composed of uppercase ASCII characters with the logical words separated by an underscore "_" (for example, "WM_ICON_NAME"). 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, then special quoting conventions would have to be used.

1.2.4 Semantics

The core protocol imposes no semantics on atoms except as they are used in FONTPROP structures. See the definition of **QueryFont** in

the protocol specification for more information on FONTPROP semantics.

1.2.5 Name Spaces

The protocol defines six distinct spaces in which Atoms are interpreted, as shown in Table 1. Any particular Atom may or may not have some valid interpretation with respect to each of these name spaces.

Table 1. Atom Name Spaces

Space	*aka*	*Examples*
Property name	name	(WM_HINTS, WM_NAME, RGB_BEST_MAP, etc.)
Property type	type	(WM_HINTS, CURSOR, RGB_COLOR_MAP, etc.)
Selection name	selection	(PRIMARY, SECONDARY, CLIPBOARD)
Selection target	target	(FILE_NAME, POSTSCRIPT, PIXMAP, etc.)
Font property		(QUAD_WIDTH, POINT_SIZE, etc.)
ClientMessage type		(WM_SAVE_YOURSELF, _DEC_SAVE_EDITS, etc.)

2. PEER-TO-PEER COMMUNICATION VIA SELECTIONS

The primary mechanism X11 defines for clients that want to exchange information, for example by cutting and pasting between windows, is *selections*. There can be an arbitrary number of selections, each named by an atom, and they are global to the server. The choice of an atom to be used is discussed in section 2.6. 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 name of the selection
- The name of a property
- A window
- An atom representing the datatype required

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

- Convert the contents of the selection to the requested datatype
- 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 it and convert it into a string is definitely antisocial.

Note that, in the X11 environment, *all* data transferred between an owner and a requestor must normally go via the server. An X11 client cannot assume that another client can open the same files, or even communicate directly. The other client may be talking to the server via 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, hostnames and port numbers, and so on, is permitted only if both clients specifically agree.

2.1 Acquiring Selection Ownership

A client wishing to acquire ownership of a particular selection should call **SetSelectionOwner**:

```
SetSelectionOwner
        selection:      ATOM
        owner:          WINDOW or None
        time:           TIMESTAMP or CurrentTime
```

The client should set "selection" to the Atom representing the selection, set "owner" to some window that it created, and set "time" to some time between the current last-change time of the selection concerned and the current server time. This time value will normally be obtained from the timestamp of the event triggering the acquisition of the selection. Clients should *not* set the time value to **CurrentTime**, since if they do so they have no way of finding when they gained ownership of the selection. Clients must use a window they created in order for requestors to be able to route events to the owner of the selection.[2]

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.

Note that if the time in the **SetSelectionOwner** request is in the future relative to the server's current time, or if it is in the past relative to the last time the selection concerned changed hands, the **SetSelectionOwner** request appears to the client to succeed, but ownership is *not* actually transferred.

---

2. There is at present no part of the protocol that requires requestors to send events to the owner of a selection. This restriction is imposed in order to prepare for possible future extensions.

Since clients cannot name other clients directly, the "owner" window is used to refer to the owning client in the replies to **GetSelectionOwner**, and 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
        selection:      ATOM
=>
        owner:          WINDOW or None
```

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

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

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

Problem: There is no way for anyone to find out the last-change time of a selection. At the next protocol revision, **GetSelectionOwner** should be changed to return the last-change time as well as the owner.

2.2 Responsibilities of the Selection Owner

When a requestor wants the value of a selection, the owner receives a **SelectionRequest** event:

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

The owner and the selection fields will be the values 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**, using **SendEvent** 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 "property" field is **None**, the requestor is an obsolete client. Owners are encouraged to support these clients by using the "target" atom as the property name to be used for the reply. Otherwise, the owner should use the "target" field to decide the form to convert the selection into, and if the selection cannot be converted into that form, refuse the **SelectionRequest** similarly.

If the "property" field is not **None**, the owner should place the data resulting from converting the selection into the specified property on the requestor window, setting the property's type to some appropriate value (which need not be the same as "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 above. See the section on "Large Data Transfers" below.

If the property is successfully stored, the owner should acknowledge the successful conversion by sending the requestor window a **SelectionNotify** event, using **SendEvent** with an empty mask:

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

The "selection," "target," "time," and "property" fields of the **SelectionNotify** event should be set to the values received in the **SelectionRequest** event (setting the "property" field to **None** indicates that the conversion requested could not be made).

Convention: The "selection," "target," "time," and "property" fields in the **SelectionNotify** event should be set to the values received in the **SelectionRequest** event.

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 section 2.4) 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). They should express interest in **PropertyNotify** events for the "requestor" window and wait until the property in the **SelectionNotify** event has been deleted before assuming that the selection data has been transferred.

When some other client acquires a selection, the previous owner receives a **SelectionClear** event:

```
SelectionClear
        owner:          WINDOW
        selection:      ATOM
        time:           TIMESTAMP
```

The "timestamp" field is the time at which the ownership changed hands, and the "owner" field is the window the new owner specified in its **SetSelectionOwner** request.

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

2.3 Giving Up Selection Ownership

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

2.3.1 Voluntarily

To relinquish ownership of a selection voluntarily, a client should execute a **SetSelectionOwner** request for that selection atom, with "owner" specified as **None**, and "time" 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 it may terminate. In both cases the ownership of the selection involved will revert to **None**.

2.3.2 Forcibly

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 **SelectionClear** event:

```
SelectionClear
        owner:          WINDOW
        selection:      ATOM
        time:           TIMESTAMP
```

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

2.4 Requesting a Selection

A client wishing to obtain the value of a selection in a particular form issues a **ConvertSelection** request:

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

The selection field specifies the particular selection involved, and the target specifies the required form of the information. The choice of suitable atoms to use is discussed in section 2.6. The requestor field should be set to a window the requestor created; the owner will place the reply property on it. The time field should be set to the timestamp on the event triggering the request for the selection value; clients should *not* use **CurrentTime** for this field.

Convention: Clients should not use **CurrentTime** for the time field of **ConvertSelection** requests. They should use the timestamp of the event that caused the request to be made.

The property field should be set to the name of a property that the owner can use to report the value of the selection. Note that the requestor of a selection needs to know neither the client owning the selection nor the window it is attached to.

Although the protocol allows the property field to be set to **None** (in which case the owner is supposed to choose a property name), it is difficult for the owner to do so safely.

Convention: Requestors should not use **None** for the property field of **ConvertSelection** requests.

Convention: Owners receiving **ConvertSelection** requests with property field **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:

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

The "requestor," "selection," "time," and "target" fields will be the same as those on the **ConvertSelection** request.

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

If the "property" field 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:

```
GetProperty
        window:         WINDOW
        property:       ATOM
        type:           ATOM or AnyPropertyType
        long-offset:    CARD32
        long-length:    CARD32
        delete:         BOOL
=>
        type:           ATOM or None
        format:         {0, 8, 16, 32}
        bytes-after:    CARD32
        value:          LISTofINT8 or LISTofINT16 or LISTofINT32
```

When using **GetProperty** to retrieve the value of a selection, the "property" field should be set to the corresponding value in the **SelectionNotify** event. The "type" field should be set to **AnyPropertyType**, because the requestor has no way of knowing beforehand what type the selection owner will use. Several **GetProperty** requests may be needed to retrieve all the data in the selection; each should set the "long-offset" field to the amount of data received so far, and the "size" field to some reasonable buffer size (see the section on "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 the section on "Selection Properties"), the property in the **SelectionNotify** should be deleted by invoking **GetProperty** with the "delete" field set True. As discussed above, 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. They should invoke either **DeleteProperty** or **GetWindowProperty(delete==TRUE)** after they have successfully retrieved all data comprising the selection. See the next section.

2.5 Large Data Transfers

Selections can get large, and this 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:

Convention: Selection owners should transfer the data describing a selection large compared with maximum-request-size in the connection handshake using the INCR property mechanism (see below).

Convention: 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 **XSetErrorHandler()** to override the default handler.

Convention: A selection owner must confirm that no Alloc error occurred while storing the properties for a selection before replying with a confirming **SelectionNotify** event.

Convention: When storing large amounts (relative to max-request-size) of data, clients should use a sequence of **ChangeProperty(mode==Append)** requests for reasonable quantities of data. This is to avoid locking-up

servers, and to limit the waste of data transfer caused by an Alloc error.

Convention: If an Alloc error occurs during storing the selection data, all properties stored for this selection should be deleted, and the **ConvertSelection** request refused by replying with a **SelectionNotify** event with "property" set to **None**.

Convention: In order to avoid locking-up servers 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.

Problem: Single-threaded servers should be changed to avoid locking-up during large data transfers.

2.6 Usage of Selection Atoms

It is important to observe that defining a new atom consumes resources in the server, and they are not released until the server reinitializes. Thus, it must be a goal to reduce the need for newly minted atoms.

2.6.1 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.

Problem: How does a client find out which selection atoms are valid?

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

2.6.1.2 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 wish to disturb it.

2.6.1.3 The CLIPBOARD Selection. The selection named by the atom CLIPBOARD is used to hold data being transferred between clients, normally being "cut" 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 normal way, retaining the data in order 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 nor provide feedback suggesting that it has actually transferred data to the clipboard.

This process should be repeated whenever the data to be transferred would change.

Clients wishing 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 data, the owner of the CLIPBOARD selection may be a single, special client implemented for the purpose. It should:

- Assert ownership of the CLIPBOARD selection, and reassert it any time the clipboard data changes.
- If it loses the selection (which will occur because someone has some new data for the clipboard):
  - — Obtain the contents of the selection from the new owner, using the timestamp in the **SelectionClear** event.
  - — Attempt to reassert ownership of the CLIPBOARD selection, using the same timestamp.

— If the attempt fails, restart the process using a newly acquired timestamp. 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 normal way, for example by a zero-length append to a property.

- Respond to requests for the CLIPBOARD contents in the normal 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, reducing the chance of race conditions causing problems.

The reasons not to run the clipboard client include:

- Performance – data is only transferred if it is actually required (when some client actually wants the data).
- Flexibility – the clipboard data may be available as more than one target.

2.6.2 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, Table 2 contains those that have been suggested so far.

Table 2. Initial Set of Target Atoms and Their Meanings

Atom	*Type (see Table 3)*	*Meaning*
TARGETS	ATOM	List of valid target atoms
MULTIPLE	ATOM_PAIR	Look in the ConvertSelection property
TIMESTAMP	INTEGER	Timestamp used to acquire selection
STRING	STRING	ISO Latin 1 (+TAB+NEWLINE) text
TEXT	TEXT	Text in owner's encoding
LIST_LENGTH	INTEGER	Number of disjoint parts of selection
PIXMAP	DRAWABLE	Pixmap ID
DRAWABLE	DRAWABLE	Drawable ID
BITMAP	BITMAP	Bitmap ID
FOREGROUND	PIXEL	Pixel value
BACKGROUND	PIXEL	Pixel value
COLORMAP	COLORMAP	Colormap ID
ODIF	TEXT	ISO Office Document Interchange Format
OWNER_OS	TEXT	Operating system of owner
FILE_NAME	TEXT	Full path name of a file
HOST_NAME	TEXT	See WM_CLIENT_MACHINE
CHARACTER_POSITION	SPAN	Start and end of selection in bytes
LINE_NUMBER	SPAN	Start and end line numbers
COLUMN_NUMBER	SPAN	
LENGTH	INTEGER	Number of bytes in selection
USER	TEXT	Name of user running owner
PROCEDURE	TEXT	Name of selected procedure
MODULE	TEXT	Name of selected module
PROCESS	INTEGER, TEXT	Process ID of owner
TASK	INTEGER, TEXT	Task ID of owner
CLASS	TEXT	Class of owner – see WM_CLASS
NAME	TEXT	Name of owner – see WM_NAME
CLIENT_WINDOW	WINDOW	Top-level window of owner
DELETE	NULL	True if owner deleted selection
INSERT_SELECTION	NULL	Insert specified selection
INSERT_PROPERTY	NULL	Insert specified property
		This table will grow.

Selection owners are required to support the following targets:

TARGETS — The owner should return a list of Atoms representing 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 field 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.

— The owner should replace in the MULTIPLE property any property atoms for targets it failed to convert with **None**.

Convention: The entries in a MULTIPLE property must be processed in the order they appear in the property. See section 2.6.3.

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 **GetSelectionOwner** returns the timestamp used to acquire ownership, selection owners must support conversion to TIMESTAMP, returning the timestamp they used to obtain the selection.

Problem: The protocol should be changed to return, in response to a **GetSelectionOwner**, the timestamp used to acquire the selection.

All other targets are optional.

2.6.3 Selection With Side Effects

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

In general, targets with side effects will return no information (i.e., a zero-length property of type NULL). 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.

Convention: Targets with side effects should return no information (i.e., a zero-length property of type NULL).

Convention: The side effect of a target must be performed before the conversion is accepted.

Convention: 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."

2.6.3.1 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.

2.6.3.2 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 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).

2.6.3.3 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).

2.7 Usage 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 type of the property involved is always chosen by the selection owner and can involve some types with special semantics assigned by convention. These special types are reviewed in the following sections.

In all cases, a request for conversion to a target should return a property of one of the types listed in Table 2 for that property, or a property of type INCR and then a property of one of the listed types.

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, side-effect targets are examples, will be of length 0 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; for variable length items such as text, the separators are listed in Table 3.

Table 3 – Property Types, Formats, and Separators		
Type Atom	*Format*	*Separator*
STRING	8	Null
ATOM	32	Fixed-size
ATOM_PAIR	32	Fixed-size
BITMAP	32	Fixed-size
PIXMAP	32	Fixed-size
DRAWABLE	32	Fixed-size
SPAN	32	Fixed-size
INTEGER	32	Fixed-size
WINDOW	32	Fixed-size
INCR	32	Fixed-size
		This table will grow.

2.7.1 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 Table 2, the word TEXT 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.

Type STRING properties will consist of a list of elements separated by NULL characters; other encodings will need to specify an appropriate list format.

2.7.2 INCR Properties

Requestors may receive a property of type INCR[3] 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, representing a lower bound on the number of bytes of data in the selection. The requestor and the selection owner transfer the data comprising 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.
- Between each append, waits for a **PropertyNotify(state==Deleted)** event showing that the requestor has read the data. The reason for doing this is to limit the consumption of space in the server.
- When the entire data has been transferred to the server, waits until a **PropertyNotify(state==Deleted)** showing 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:

---

3. 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.

  - — Retrieving data using **GetProperty** with "delete" True
  - — Waiting for a **PropertyNotify** with state==NewValue
- Until a zero-length property is obtained
- 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.

2.7.3 DRAWABLE Properties

Requestors may receive properties of type PIXMAP, BITMAP, DRAWABLE, or WINDOW, containing an appropriate ID. Some information about these drawables is available from the server via the **GetGeometry** request, but the following items are not:

- Foreground pixel
- Background pixel
- Colormap ID

In general, requestors converting into targets whose returned type in Table 2 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

2.7.4 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 the ending position plus one. The base is zero. If they are the same, the span is zero length, and before the specified position. The units are implied by the target atom, such as LINE_NUMBER or CHARACTER_POSITION.

3. PEER-TO-PEER COMMUNICATION VIA CUT-BUFFERS

Communication via cut-buffers is much simpler, but much less powerful than via 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, where a requestor retrieves it at some later time.

The cut-buffers consist of eight properties on the root of screen 0, named by the predefined atoms CUT_BUFFER0 to CUT_BUFFER7. These properties must (at present) have type STRING and format 8. A client using the cut-buffer mechanism must initially ensure that all eight exist, using **ChangeProperty** to append zero-length data to each.

A client storing data in the cut-buffers (an owner) must first rotate the ring of buffers by +1, using **RotateProperties** to rename CUT_BUFFER0 to CUT_BUFFER1 to ..... to CUT_BUFFER7 to CUT_BUFFER0. It must then store the data into CUT_BUFFER0, using **ChangeProperty** in mode Replace.

A client obtaining data from the cut-buffers should use **GetProperty** to retrieve the contents of CUT_BUFFER0.

A client may, in response to a specific user request, rotate the cut-buffers by −1, using **RotateProperties** to rename CUT_BUFFER7 to CUT_BUFFER6 to ..... to 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.

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 and again from the window manager's point of view in the *Window and Session Manager Conventions Manual*.[4]

In general, these conventions are somewhat complex and will undoubtedly change over time as new window management paradigms are developed. Thus there is a strong bias towards 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 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; it merely ensures that no claim of general utility is made for such programs.

---

4. The *Window and Session Manager Conventions Manual* will be prepared after this manual is finalized.

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 (e.g., the debugger) to appear on top.

4.1 Client's Actions

In general, the object of the X11 design is that clients should, as far as possible, do exactly what they would do in the absence of a window manager, except for:

- 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.

4.1.1 Creating a Top-Level Window

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

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

or, if a particular one of the roots was required, like:

```
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, 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.[5]

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 occur when the top-level window is mapped and unmapped and when the window manager receives certain messages. For details, see sections 4.1.2.4 and 4.1.4.

4.1.2 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 its desired behavior. 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. Properties written by the client will not be changed by the window manager.

The window manager will examine the contents of these properties when the window makes the transition from Withdrawn state, and will monitor some for changes while the window is in 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 is to ensure

5. The mechanism proposed in the earlier drafts turned out to be inadequate to support all the proposed uses of the pseudo-root facility.

that the full contents of the property will be available to a new window manager if the existing one crashes, or 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 the property remains valid at all times.

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 following sections describe each of the properties the clients need to set in turn. They are summarized in Table 13 in section 4.3.

4.1.2.1 WM_NAME. The WM_NAME property is an uninterpreted string that the client wishes 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 ATOMS to be used for this purpose are described in section 2.7.1.

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 made an explicit decision 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 offscreen, or have covered it by other windows. WM_NAME should not be used for application-critical information, nor to announce asynchronous changes of application 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
- To provide the user with noncritical state information

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

Problem: A change to **XFetchName** and similar Xlib routines is needed to allow for multiple encodings.

4.1.2.2 WM_ICON_NAME. The WM_ICON_NAME property is an uninterpreted string that the client wishes 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. Fewer characters will normally be visible in WM_ICON_NAME than WM_NAME, for obvious geometric reasons.

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

4.1.2.3 WM_NORMAL_HINTS. The type of the WM_NORMAL_HINTS property is WM_SIZE_HINTS. Its contents are shown in Table 4.

Table 4. WM_SIZE_HINTS Type Property Contents

Field	*Type*	*Comments*
flags	CARD32	See Table 5 below
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

Table 5. WM_SIZE_HINTS.flags Bit Definitions		
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
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 mapped from Withdrawn state) was specified by the user, the client should set the **USPosition** and **USSize** flags. To indicate that it was specified by the client without any user involvement, the client should set **PPosition** and **PSize**. **USPosition** and **USSize** 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.

The size specifiers refer to the width and height of the client's window excluding borders. The window manager will interpret the position of the window, and its border width, to position the point of the outer rectangle of the overall window specified by the win_gravity in the size hints. The outer rectangle of the window includes any borders or decorations supplied by the window manager. In other words, if the window manager decides to place the window where the client asked, the position on the parent window's border named by the win_gravity will be placed where the client window would have been placed in the absence of a window manager.

The defined values for win_gravity are those specified for WINGRAVITY in the core X protocol, with the exception of Unmap and Static: NorthWest (1), North (2), NorthEast (3), West (4), Center (5), East (6), SouthWest (7), South (8), and SouthEast (9).

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:

```
width = base_width + ( i * width_inc )
height = base_height + ( j * height_inc )
```

for non-negative integers i and j. 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.

Problem: The "base" and "win_gravity" fields need a change to Xlib.

4.1.2.4 WM_HINTS. The WM_HINTS property, whose type is WM_HINTS, is used to communicate to the window manager the information it needs other than the window geometry, which is available from the window itself, the constraints on that geometry, which are available from the WM_NORMAL_HINTS structure, and various strings, which need separate properties such as WM_NAME. The contents of these properties are shown in Table 6.

Table 6. WM_HINTS Type Property Contents

Field	*Type*	*Comments*
flags	CARD32	See Table 7 below
input	CARD32	Client's input model
initial_state	CARD32	State when first mapped
icon_pixmap	PIXMAP	Pixmap for icon image
icon_window	WINDOW	Window for icon image
icon_x	INT32	Icon location
icon_y	INT32	
icon_mask	PIXMAP	Mask for icon shape
window_group	WINDOW	ID of group leader window

Table 7. WM_HINTS.flags Bit Definitions		
Name	*Value*	*Field*
InputHint	1	input
StateHint	2	initial_state
IconPixmapHint	4	icon_pixmap
IconWindowHint	8	icon_window
IconPositionHint	16	icon_x and icon_y
IconMaskHint	32	icon_mask
WindowGroupHint	64	window_group
MessageHint	128	This bit is obsolete

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 section 4.1.7.).

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 section 4.1.4. Newly created windows start in the Withdrawn state. Transitions between states happen when a non-override-redirect 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 Withdrawn state, as shown in Table 8.

Table 8. WM_HINTS.initial_state Values		
State	*Value*	*Comments*
NormalState	1	Window is visible
IconicState	3	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 section 4.1.3.2).
- 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, so that clients should not attempt to remedy the omission by working around it.

Clients needing more capabilities from the icons than a simple two-color bitmap should use icon windows. Rules for clients that do are set out in section 4.1.9.

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.

Convention: The window_group field should be set to the ID of the group leader. The window group leader may be a window which 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.

Convention: 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,[6] rather than the WM_PROTOCOLS mechanism of section 4.1.2.7.

4.1.2.5 WM_CLASS. The WM_CLASS property, of type STRING (without control characters), contains two consecutive null-terminated strings specifying 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 Withdrawn state, and may be changed only while the window is in Withdrawn state. Window managers may examine the property only when they start up and when the Window leaves Withdrawn state, but there should be no need for a client to change its state dynamically.

The two strings are, respectively:

- A string naming the particular instance of the application to which the client owning this window belongs. Resources that are specified by instance name override any resources that are specified by class name. Instance names may be specified by the user in an operating system-specific manner. Under the UNIX* System, 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.

6. This obsolete protocol was described in the 27th July 1988 draft of this manual. 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.

— 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 naming the general class of applications to which the client owning 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," etc.

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

4.1.2.6 WM_TRANSIENT_FOR. 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 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 popup menus), use override-redirect and grab the pointer while the window is mapped.

4.1.2.7 WM_PROTOCOLS. 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, as well as 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 section 4.2.8. 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 wish to participate in any window manager protocols.

The X Consortium will maintain a registry of protocols to avoid collisions in the name space. Table 9 contains the protocols that have been defined to date.

Table 9. Current WM_PROTOCOLS

Protocol	*Section*	*Purpose*
WM_TAKE_FOCUS	4.1.7	Assignment of input focus
WM_SAVE_YOURSELF	5.2.1	Save client state warning
WM_DELETE_WINDOW	5.2.2	Request to delete top-level window
		This table will grow.

4.1.2.8 WM_COLORMAP_WINDOWS. 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.

See section 4.1.8 for the details of this mechanism.

4.1.3 Window Manager Properties

The properties described above are those which 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.

4.1.3.1 WM_STATE. The window manager will place a WM_STATE property, of type WM_STATE, on each top-level client window. In general, clients should not need to examine the contents

of this property; it is intended for communication between window and session managers. See section 5.1.1.3 for more details.

4.1.3.2 WM_ICON_SIZE. 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 shown in Table 10.

Table 10. WM_ICON_SIZE Type Property Contents

Field	*Type*	*Comments*
min_width	CARD32	Data for icon size series
min_height	CARD32	
max_width	CARD32	
max_height	CARD32	
width_inc	CARD32	
height_inc	CARD32	

For more details see the *Xlib Programming Manual, Volume One* and the *Xlib Reference Manual, Volume Two.*

4.1.4 Changing Window State

From the client's point of view, the window manager will regard each of the client's top-level non-override-redirect windows as being in one of three states. The semantics of the states are:

- **NormalState**
  The client's top-level window is visible.
- **IconicState**
  The client's top-level window is iconic, whatever that means for this window manager. The client can assume that its icon_window (if any) will be visible, and failing that, its icon_pixmap (if any) or its WM_ICON_NAME will be visible.
- **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 **InactiveState** 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 **IconicState**.

Newly created top-level windows are in Withdrawn state. Once the window has been provided with suitable properties, the client is free to change its state as follows:[7]

- 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 client message event as described below.
- Normal → Withdrawn
  The client should unmap the window and follow it with a synthetic **UnmapNotify** event as described below.[8]
- 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 below.

Once a client's non-override-redirect top-level window has left Withdrawn state, the client will know that the window is in Normal state if it is mapped, and that the window is in Iconic state if it is not mapped. It may select for **StructureNotify** on the top-level window, and it will receive an **UnmapNotify** event when it moves to Iconic state and a **MapNotify** when it moves to Normal state. This implies

---

7. The conventions described in earlier drafts of this manual had some serious semantic problems. These new conventions are designed to be compatible with clients using earlier conventions, except in areas where the earlier conventions would not actually have worked.

8. For compatibility with obsolete clients, window managers should trigger the transition on the real **UnmapNotify** rather than wait 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**.

that a reparenting window manager will unmap the top-level window as well as the parent window when going Iconic.

Convention: Reparenting window managers must unmap the client's top-level window whenever they unmap the window to which they have reparented it.

If the transition is to Withdrawn state, in addition to unmapping the window itself, a synthetic **UnmapNotify** event must be sent using **SendEvent** with the following parameters:

```
destination:            the root
propagate:              False
event-mask:             (SubstructureRedirect|SubstructureNotify)
event: an UnmapNotify with:
       event:           the root
       window:          the window itself
       from-configure: False
```

The reason for doing this is to ensure that the window manager gets some notification of the desire to change state, even though the window may already be unmapped when the desire is expressed.

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

- "window" == the window to be iconified
- "type" == the atom WM_CHANGE_STATE[9]
- "format" == 32
- "data[0]" == IconicState

Other values of data[0] are reserved for future extensions to these conventions.[10] The parameters of the **SendEvent** should be as above.

Clients can also select for **VisibilityChange** on their (top-level or icon) windows. They will then receive a **VisibilityNotify(state=FullyObscured)** event when the window

---

9. 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**).

10. The format of this **ClientMessage** event does not match the format of **ClientMessages** in section 4.2.8. This is because they are sent by the window manager to clients, and this is sent by clients to the window manager.

concerned becomes completely obscured even though mapped (and thus perhaps a waste of time to update), and a **VisibilityNotify(state!=FullyObscured)** when it becomes even partly viewable.

4.1.5 Configuring the Window

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

- 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, and 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 Withdrawn state, as described in section 4.1.2.3. 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** by:

- Not changing the size or location of the window at all, a client will receive a synthetic **ConfigureNotify** event describing the (unchanged) state 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**, since no change has actually taken place.
- Moving the window without resizing it, a client will receive a synthetic **ConfigureNotify** event following the move describing the new state of the window, whose (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 describing this change, since the window manager may have re-parented the top-level window. If it does receive a real event, the synthetic event will follow the real one.

- Resizing the window (whether or not it is moved), a client which has selected for **StructureNotify** will receive a **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, and 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, coordinates in real **ConfigureNotify** events are in the parent's space, whereas in synthetic events they are in the root space.

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 should, therefore, not depend on the top-level window's border being visible nor use it to display any critical information. Other window managers will allow the top-level windows border to be visible.

Convention: Clients should set their desired border-width on all **ConfigureWindow** requests, to avoid a race condition.

Clients changing 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 using **ConfigureWindow** 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 **SendEvent** with:

```
destination:    the root
propagate:      False
event-mask:     (SubstructureRedirect|SubstructureNotify)
event:  a ConfigureRequest with:
        event:  the root
        window: the window itself
        ....    other parameters from the ConfigureWindow
```

Doing this is deprecated, and window managers are in any case free to position windows in the stack as they see fit. Clients should ignore the "above" fields of both real and synthetic **ConfigureNotify** events that they receive on their non-override-redirect top-level windows since they cannot be guaranteed to contain useful information.

4.1.6 Changing Window Attributes

The attributes that may be supplied when a window is created may be changed using the **ChangeWindowAttributes** request. They are shown in Table 11.

Table 11. Window Attributes

Attribute	*Private to Client*
Background pixmap	Yes
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 are private to the client and will never be interfered with by the window manager. As regards 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, 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 sections 4.1.10 and 4.2.9.

4.1.7 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 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, once 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 shown in Table 12.

Table 12. Input Models

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 **True** to indicate that they require window manager assistance in acquiring the input focus. No Input and Globally Active clients set the "input" field **False** to request that the window manager not set the input focus to their top-level windows.

Clients using **SetInputFocus** must set the "time" field to the time-stamp of the event that caused them to make the attempt. Note that this cannot be a **FocusIn** event, since they do not have time-stamps, and that clients may acquire the focus without a corresponding **EnterNotify**. Clients must not use **CurrentTime** in the "time" field.

Clients using the Globally Active model can only use **SetInputFocus** 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** from the window manager as described in section 4.2.8 with WM_TAKE_FOCUS in their data[0] 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. See section 4.2.7.

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 can either assign it to one of its subwindows or decline the offer of the focus. A clock, for example, 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.

Problem: 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.

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. It can be any of:

- **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, since 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 (most window managers will) and then crashes. The input focus will be **None**, and there will probably be no way to change it.

The convention is:

Convention: Clients invoking **SetInputFocus** should set "revert-to" to Parent.

A convention is also required for clients that want to give up the input focus.

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

4.1.8 Colormaps

The window manager is responsible for installing and uninstalling colormaps.[11] Clients provide the window manager with hints on which colormaps to install and uninstall, but must not install or uninstall colormaps themselves. 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 insure that one or more of the client's colormaps are installed. The reason for this convention is that there is no safe way for multiple clients to install and uninstall Colormaps.

Convention: Clients must not use **InstallColormap** or **UninstallColormap**.

There are two possible ways in which clients could hint to the window manager about the Colormaps they want installed. Using a property, they could tell the window manager:

- a priority ordered list of the Colormaps they want installed
- or a priority ordered list of the Windows whose Colormaps they want installed.

The second of these alternatives has been selected because:

- It allows window managers to know the Visuals for the Colormaps, permitting Visual-dependent colormap installation policies.
- It allows window managers to select for **VisibilityChange** on the windows concerned and ensure that maps are only installed if the windows that need them are visible.

Clients whose top-level windows and subwindows all use the same colormap should set their IDs in the colormap field of the 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 window attribute, and the window manager will install the colormap for them.

---

11. The conventions described in earlier drafts by which clients and window managers shared responsibility for installing Colormaps suffered from semantic problems.

Clients creating windows may use the value **CopyFromParent** to inherit the 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 window IDs of 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 section 4.1.2.8). The list is ordered by the importance to the client of having the colormaps installed. If this order changes, the property should be updated. The window manager will track changes to this property, and will track changes to the colormap attribute of the windows in the property.

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.

Clients should be aware of the min-installed-maps and max-installed-maps fields of the connection startup information, and the effect that the minimum value has on the "required list":

> "At any time, there is a subset of the installed maps, viewed as an ordered list, called the "required list". The length of the required list is at most M, where M is the min-installed-maps specified for the screen in the connection setup. The required list is maintained as follows. When a colormap is an explicit argument to **InstallColormap**, it is added to the head of the list, and the list is truncated at the tail if necessary to keep the length of the list to at most M. When a colormap is an explicit argument to **UninstallColormap** and it is in the required list, it is removed from the list. A colormap is not added to the required list when it is installed implicitly by the server, and the server cannot implicitly uninstall a colormap that is in the required list."

In other words, the min-installed-maps most recently installed maps are guaranteed to be installed. This number will often be one; clients needing multiple colormaps should beware.

The window manager will identify and track changes to the colormap attribute of the windows identified by the WM_COLORMAP_WINDOWS property, and the top-level window if it does not appear in the list. If the top-level window does not appear in the list, it will be assumed to be a higher priority than any window in the list. It will also track changes in the contents of the WM_COLORMAP_WINDOWS property, in case the set of windows or their relative priority changes. The window manager will define some colormap focus policy, and whenever the top-level window has the colormap focus, it will attempt to maximize the number of colormaps from the head of the WM_COLORMAP_WINDOWS list that are installed.

4.1.9 Icons

A client can hint to the window manager about the desired appearance of its icon in several ways:

- Set a string in WM_ICON_NAME. All clients should do this, as it provides a fallback for window managers whose ideas about icons differ widely from those of the client.
- Set 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 will normally clip or tile pixmaps which do not match WM_ICON_SIZE.
- Set 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 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 Iconic state, the window manager will normally ensure that:

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

Clients should observe the following conventions about their icon windows:

Convention: The icon window should be an InputOutput child of the root.

Convention: The icon window should be one of the sizes specified in the WM_ICON_SIZE property on the root.

Convention: The icon window should use the root visual and default colormap for the screen in question.

Convention: Clients should not map their icon windows.

Convention: Clients should not unmap their icon windows.

Convention: Clients should not configure their icon windows.

Convention: Clients should not set override-redirect on their icon windows, nor select for **ResizeRedirect** on them.

Convention: Clients must not depend on being able to receive input events via their icon windows.

Convention: Clients must not manipulate the borders of their icon windows.

Convention: Clients must select for Exposure 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 some subset of the keys and buttons though.

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

4.1.10 Pop-Up Windows

Clients wishing 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 below.
- 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.

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.

4.1.11 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 (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 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. There is, at present, no way for a client to request a group, as opposed to an individual, operation.

4.2 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.

4.2.1 Reparenting

Clients must be aware that some window managers will reparent their non-override-redirect 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:

- 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 normally 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 wishing 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. They will, in fact, normally not be interpreted by the server because a reparenting window manager will normally have intercepted these operations (see below). Clients should use the root coordinate space for these requests (see section 4.1.5).
- **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 section 4.1.5).
- 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 wishing to be notified when they are reparented can select for **StructureNotify** on their top-level window. They will receive a **ReparentNotify** event if and when reparenting takes place.

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 if the window manager terminates, the reparented window will not be destroyed, 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.

When the window manager gives up control over a client's top-level window, it will reparent it (and any associated windows, such as WM_TRANSIENT_FOR windows) back to the root.

There is a potential race condition here. A client might wish to reuse the top-level window, reparenting it somewhere else.

Convention: Clients wishing to reparent their top-level windows should do so only when they have their original parents. They may select for **StructureNotify** on their top-level windows, and will receive **ReparentNotify** events informing them when this is true.

4.2.2 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 may not assume that any redirectable request is actually performed when the request is issued, or at all. For example, the sequence:

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

is incorrect, since the **MapWindow** request may be intercepted and the **PolyLine** output made to an unmapped window. The client must wait for an **Expose** event before drawing in the window.[12] Another example is:

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

which incorrectly assumes that the **ConfigureWindow** request is actually executed with the arguments supplied.

The requests which may be redirected are:

- **MapWindow**
- **ConfigureWindow**
- **CirculateWindow**

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 section 4.1.10) and while responding to **ResizeRequest** events (see section 4.2.9).

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 pre-existing window manager from interfering with their activities.
- Restore the status quo exactly after they unmap the window, so that any pre-existing window manager does not get confused.

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

---

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

4.2.3 Window Move

If the window manager moves a top-level window without changing its size, the client will receive a synthetic **ConfigureNotify** event describing 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. And note that the coordinates in the event will not, in this case, be directly useful.

The window manager will send these events using **SendEvent** with:

```
destination:    the client's window
propagate:      False
event-mask:     StructureNotify
```

4.2.4 Window Resize

The client can elect to receive notification of being resized by selecting for **StructureNotify** on its top-level window(s). 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 Iconic state.

4.2.5 (De)Iconify

A non-override-redirect window that is not Withdrawn will be in Normal state if it is mapped, and in 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 Iconic state.

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

4.2.6 Colormap Change

Clients that wish to be notified of their colormaps being installed or uninstalled should select for **ColormapNotify** 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.

Problem: There is an inadequacy in the protocol. At the next revision, the **InstallColormap** request should be changed to include a timestamp to avoid the possibility of race conditions if more than one client attempts to install and uninstall colormaps. These conventions attempt to avoid the problem by restricting use of these requests to the window manager.

4.2.7 Input Focus

Clients can request notification that they have the input focus by selecting for **FocusChange** 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 set the "input" field of WM_HINTS to True and actually have the input focus in (one of) their top-level windows,
- or have set the "input" field of WM_HINTS to False and have received a suitable event as described in section 4.1.7,
- or have received a WM_TAKE_FOCUS message as described in section 4.1.7.

Clients should not warp the pointer in an attempt to transfer the focus; they should set the focus and leave the pointer alone. See section 6.2.

Once a client satisfies these conditions, it may transfer the focus to another of its windows using the **SetInputFocus** request:

```
SetInputFocus
        focus:          WINDOW or PointerRoot or None
        revert-to:      {Parent, PointerRoot, None}
        time:           TIMESTAMP or CurrentTime
```

Convention: Clients using **SetInputFocus** must set the "time" field to the timestamp of the event that caused them to make the attempt. Note that this cannot be a **FocusIn** event (since they do not have timestamps) and that clients may acquire the focus without a corresponding **EnterNotify**. Clients must not use **CurrentTime** in the "time" field.

Convention: Clients using **SetInputFocus** to set the focus to one of their windows must set the revert-to field to **Parent**.

4.2.8 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 section 4.1.2.7). For all protocols, the **ClientMessage** events:

- Have WM_PROTOCOLS as the type field[13]
- Have format 32
- Have the atom naming their protocol in the data[0] field[14]
- Have a timestamp in their data[1] field

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

These events will be sent using **SendEvent** with:

```
destination:    the client's window
propagate:      False
event-mask:     () empty
event:          as specified by the protocol
```

13. 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**).

14. 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.

4.2.9 Redirecting Requests

Normal clients can use the redirection mechanism just as window managers do, by selecting for **SubstructureRedirect** on a parent window, or **ResizeRedirect** 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** only on windows that they own.

In particular, clients that need to take some special action if they are resized can select for **ResizeRedirect** 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: (a) setting override-redirect on the window specified in the event, (b) 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, and then (c) 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.

4.3 Summary of Window Manager Property Types

The window manager properties are summarized in Table 13.

Table 13. Window Manager Properties

Name	*Type*	*Format*	*See Section*
WM_CLASS	STRING	8	4.1.2.5
WM_COLORMAP_WINDOWS	WINDOW	32	4.1.2.8
WM_HINTS	WM_HINTS	32	4.1.2.4
WM_ICON_NAME	TEXT		4.1.2.2
WM_ICON_SIZE	WM_ICON_SIZE	32	4.1.3.2
WM_NAME	TEXT		4.1.2.1
WM_NORMAL_HINTS	WM_SIZE_HINTS	32	4.1.2.3
WM_PROTOCOLS	ATOM	32	4.1.2.7
WM_STATE	WM_STATE	32	4.1.3.1
WM_TRANSIENT_FOR	WINDOW	32	4.1.2.6

5. CLIENT TO SESSION MANAGER COMMUNICATION

The role of the session manager is to manage a collection of clients. It should be capable of:

- Starting a collection of clients as a group.
- Remembering the state of a collection of clients so that they can be restarted in the same state.
- Stopping a collection of clients in a controlled way.

It may also provide a user interface to these capabilities.

5.1 Client Actions

There are two ways in which clients should cooperate with the session manager:

- Stateful clients should cooperate with the session manager by providing it with information it can use to restart them if it should become necessary.
- Clients whose server connection needs to survive the deletion of their top-level window (typically those with more than one top-level window) should take part in the WM_DELETE_WINDOW protocol (see section 5.2.2).

5.1.1 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.

5.1.1.1 WM_COMMAND. The WM_COMMAND property represents the command used to (re)start the client. Clients should ensure, by resetting this property, that it always reflects a command that will restart them in their current state. The content and type of the property depends on the operating system of the machine running the client. In UNIX Systems using ISO Latin 1 characters for their command lines, the property should:

- be of type STRING,
- contain a list of NULL-terminated strings, and
- be initialized from *argv*. Other systems will need to set appropriate conventions for the type and contents and type of WM_COMMAND properties. Window and session managers should not assume that STRING is the type of WM_COMMAND, nor assume that they will be able to understand or display its contents.

Note that WM_COMMAND strings, being null-terminated, 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 (see section 5.2.1).

5.1.1.2 WM_CLIENT_MACHINE. The client should set the WM_CLIENT_MACHINE property, of one of the TEXT types, to a string forming the name of the machine running the client, as seen from the machine running the server.

5.1.1.3 WM_STATE. The window manager will place a WM_STATE property, of type WM_STATE, on each top-level client window.

Programs like **xprop** that want to operate on a client's top-level windows can use this property to identify them. A client's top-level window is one that:

- has override-redirect False,
- and has a WM_STATE property,
- or a mapped child of the root that has no descendant with a WM_STATE property.

Recursion is necessary to cover all window manager reparenting possibilities. Note that clients other than window and session managers should not need to examine the contents of WM_STATE properties, which are not formally defined by this document. The presence or absence of the property is all they need to know.

The suggested contents of the WM_STATE property are shown in Table 14.

Table 14. WM_STATE Type Property Contents

Field	*Type*	*Comments*
state	CARD32	See Table 15 below
icon	WINDOW	ID of icon window

Table 15. WM_STATE.state Values

State	*Value*	*Comments*
WithdrawnState	0	
NormalState	1	
IconicState	3	

Adding other fields to this property is reserved to the X Consortium.

The icon field should contain the window ID of the window that the window manager uses as the icon window for the window on which this property is set, if any; otherwise **None**. Note that this window may not be the same as the icon window that the client may have specified. It may be:

- the client's icon window,
- or a window that the window manager supplied that contains the client's icon pixmap,
- or the least ancestor of the client's icon window (or of the window which contains the client's icon pixmap) that contains no other icons.

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**, that it should animate its icon window. Note that in either state clients should be prepared to handle exposure events from either window.

The contents of WM_STATE properties and other aspects of the communication between window and session managers will be specified in the *Window and Session Manager Conventions Manual*.

5.1.2 Termination

Since they communicate via unreliable network connections, X11 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, nor on being able to use the server to negotiate with the user (for example, using dialog boxes for confirmation) about their fate.

Equally, 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.

5.2 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

5.2.1 Saving Client State

Clients that wish 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** as described in section 4.2.8. with the atom WM_SAVE_YOURSELF in its data[0] field.

Clients receiving 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** on WM_COMMAND as a confirmation that the client has saved its state, so that 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.

Clients with multiple top-level windows should ensure that, irrespective of how they are arranged in window groups:

- Only one of their top-level windows has a nonzero-length WM_COMMAND property.
- They respond to a WM_SAVE_YOURSELF message by (in this order):
  1. updating the nonzero-length WM_COMMAND property if necessary,
  2. updating the WM_COMMAND property on the window for which they received the WM_SAVE_YOURSELF message if it was not updated in step 1.

Receiving WM_SAVE_YOURSELF on a window is (conceptually) a command to save the entire client state.[15]

5.2.2 Window Deletion

Clients, normally those with multiple top-level windows, whose server connection must survive the deletion of some of their top-level

15. 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.

windows should include the atom WM_DELETE_WINDOW in the WM_PROTOCOLS property on each such window. They will receive a **ClientMessage** as described in section 4.2.8 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 dialogue with the user, and if they decide to complete the deletion:

- Either change the window's state to Withdrawn (as described in section 4.1.4) or destroy the window.
- Destroy any internal state associated with the window.

If the user aborts the deletion during the confirmation dialogue, 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 states (backing store, for example) 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** 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.

Note that the WM_SAVE_YOURSELF and WM_DELETE_WINDOW protocols are orthogonal to each other and may be selected independently.

5.3 Summary of Session Manager Property Types

The session manager properties are summarized in Table 16.

Table 16. Window Manager Properties			
Name	*Type*	*Format*	*See Section*
WM_CLIENT_MACHINE	TEXT		5.1.1.2
WM_COMMAND	TEXT		5.1.1.1
WM_STATE	WM_STATE	32	5.1.1.3

6. MANIPULATION OF SHARED RESOURCES

X11 permits clients to manipulate a number of shared resources, among them the input focus, the pointer, and colormaps. Conventions are required so that clients do so in an orderly fashion.

6.1 The Input Focus

Clients that explicitly set the input focus can do so in one of two modes:

Convention: 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 TRUE.

Convention: Globally Active clients should set the input focus to one of their windows only when they receive a button event, 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 FALSE.

Convention: 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**.

6.2 The Pointer

In general, clients should not warp the pointer. Window managers 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 wish to preserve the illusion that the user is in sole control of the pointer.

Convention: Clients should not warp the pointer.

Convention: Clients that insist on warping the pointer should do so only with the "src-window" field of the **WarpPointer** request set to one of their windows.

6.3 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 are, therefore, 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 events involving keys whose keysyms are registered as being for window management functions (e.g., 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. No keysyms are currently registered for window management functions.

Even so, clients are well-advised to allow the key and button combinations used to elicit program actions to be modified since 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 means that a window manager wishing 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, or 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 using:

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

and release the grab using **AllowEvents** with:

```
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 tradeoff here between freedom for window managers to implement their user interface policies and freedom for clients to implement theirs. We resolve this dilemma 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.

6.4 Colormaps

Convention: If a client has a top-level window that has subwindows, or override-redirect pop-up windows, whose colormap requirements differ from the top-level window should set a WM_COLORMAP_WINDOWS property on the top-level window. The WM_COLORMAP_WINDOWS property contains a list of the window IDs of windows that the window manager should track for colormap changes.

Convention: When a client's colormap requirements change, the client should change the colormap window attribute of a top-level window or one of the windows indicated by a WM_COLORMAP_WINDOWS property.

Convention: Clients must not use **InstallColormap** or **UninstallColormap**.

Clients with **DirectColor**-type applications should consult the *Xlib Programming Manual, Volume One* and the *Xlib Reference Manual, Volume Two* 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.

Note, however, that the conventions described there are not adequate if the server supports multiple Visuals and are not adequate if standard colormaps need to be deleted. To address this, two additional fields (visual_id and kill_id) are required in

RGB_COLOR_MAP-type properties, as shown in Table 17. The Colormap described by the property is one appropriate for the Screen on whose root the property is found.

Table 17. RGB_COLOR_MAP Type Property Contents

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, then the resources should be freed by issuing a **KillClient** protocol request with "kill_id" as the argument. If "kill_id" is one, then the resources should be freed by issuing a **FreeColormap** protocol request with "colormap" as the Colormap argument. If "kill_id" is zero, then no attempt should be made to free the resources. A client creating 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 creating an RGB_COLOR_MAP for which the "colormap" resource is shared in some way (e.g., 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.

A similar capability is desirable for other Visuals and can be supported by changing the definition of the RGB_DEFAULT_MAP property to read:

> "This atom names a property. The value of the property is an array of **XStandardColormap** structures (as extended to include visual_id and kill_id fields)."
>
> "Each entry in the array describes an RGB subset of the default color map for the Visual specified by visual_id."

6.5 The Keyboard Mapping

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

It is possible for clients to modify this table, using **ChangeKeyboardMapping**. 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. This conversion for ISO Latin 1 is implemented by the Xlib *XLookupString()* function; there will presumably be equivalent functions for other encodings.

The only valid reason for using **ChangeKeyboardMapping** 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.

It is permissible for a client to interact with a user in the following manner:

- "You just started me on a server without a PAUSE key. Please choose a key to be the PAUSE key and press it now."
- <User presses the SCROLL LOCK key>
- "Adding PAUSE to the symbols on the SCROLL LOCK key: Confirm or Abort."
- <User confirms>
- Client uses **ChangeKeyboardMapping** to add PAUSE to the keycode that already contains SCROLL LOCK.
- "Please paint PAUSE on the SCROLL LOCK key."

Convention: Clients should not use **ChangeKeyboardMapping**.

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.

6.6 The Modifier Mapping

X11 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**. This table is a shared resource and requires conventions.

A client needing 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 to the keycodes in its controlling set, including 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 needing to use an extra modifier, for example META, should:

- 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:
  - — If there is a keycode with XL_Meta_L in its set of keysyms, add that keycode to the set for the chosen modifier, then
  - — if there is a keycode with XL_Meta_R in its set of keysyms, add that keycode to the set for the chosen modifier, then
  - — 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.

Convention: Clients needing a modifier not currently in use should assign keycodes carrying suitable keysyms to an unused modifier bit.

Convention: 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 nonpreassigned modifiers when they are no longer being used.

Convention: The user has to use **xmodmap** or some other utility to deassign obsolete modifier mappings by hand.

Problem: This is rather "low tech."

When a client succeeds in performing a **SetModifierMapping,** all clients will receive **MappingNotify(request=Modifier)** events. There is no mechanism for preventing these events being received. A client using one of the nonpreassigned modifiers which receives one of these events should do a **GetModifierMapping** to discover the new mapping, and if the modifier it is using has been cleared, it should reinstall the modifier.

Note that **GrabServer** must be used to make the **GetModifierMapping, SetModifierMapping** pair in these transactions atomic.

7. RESOURCE MANAGER CONVENTIONS

This section has yet to be generated.

8. CONCLUSION

This document provides the protocol level specification of the minimal conventions needed to ensure that X11 clients can interoperate properly. Further documents are required:

- A *Window and Session Manager Conventions Manual* to cover these conventions from the opposite point of view and to add extra conventions of interest to window and session manager implementors.
- A addendum to the *Xlib Reference Manual, Volume Two* covering the additional routines (**XIconify**() would be an example) needed to ensure that adhering to these conventions is convenient for the C programmer.

9. ACKNOWLEDGMENTS

David Rosenthal had overall architectural responsibility for the conventions defined in this document, wrote most of the text, and edited the document, but the 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:

- For the Selection section: Jerry Farrell, Phil Karlton, Loretta Guarino Reid, Mark Manasse, and 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, and Glenn Widener.

Thanks are also due to those who contributed to the public review, including: 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, Allan Mimms, Colas Nahaboo, Mark Patrick, Steve Pitschke, Brad Reed, and John Thomas.

Appendix A: COMPATIBILITY WITH EARLIER DRAFTS

This appendix summarizes the incompatibilities between this document and earlier drafts.

1. A1: The R2 Draft

The 25 February 1988 draft that was distributed as part of X11R2 was clearly labeled as such, and many areas were explicitly labeled as liable to change. Nevertheless, in the revision work 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 earlier draft would still work.

The areas in which incompatibilities have become necessary are:

- The use of property **None** in **ConvertSelection** requests is no longer allowed. Owners receiving 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.

2. A2: The 27th July 1988 Draft

The Consortium review was based on a draft dated 27th July 1988. Incompatibilities have been introduced in the following areas:

- 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 using 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 using 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 pseudo-root 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 7 of the *Xlib Programming Manual, Volume 1* regarding properties of type RGB_COLOR_MAP have been changed, but clients using 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.

3. A3: The Public Review Drafts

The public review resulted in a set of mostly editorial changes. The changes that introduced some degree of incompatibility are:

- A new section (6.3) 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** sent by a client to iconify itself has been clarified, but there should be no incompatibility since 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 will now be ignored, but there should be no incompatibility since 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 which 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.

Appendix B: SUGGESTED PROTOCOL REVISIONS

During the development of these conventions, a number of inadequacies have been discovered in the 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. At the next protocol revision, **GetSelectionOwner** should be changed to return the last-change time as well as the owner.
- How does a client find out which selection atoms are valid?
- The protocol should be changed to return in response to a **GetSelectionOwner** the timestamp used to acquire the selection.
- 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 **InstallColormap**; the request does not take a timestamp, and it may be executed after the top-level colormap has been uninstalled. The next protocol revision should provide the timestamp in **InstallColormap**, **UninstallColormap**, **ListInstalledColormaps**, and the **ColormapNotify** event. The timestamp should be used in a similar way to the last-focus-change time for the input focus.
- 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.

INTERACTIVE TCP/IP
Programmer's Supplement

CONTENTS

1. INTRODUCTION 1
   1.1 The TCP/IP Interfaces 1
   1.2 Prerequisites 1
   1.3 Overview of This Document 2
2. THE TRANSPORT LAYER INTERFACE 3
   2.1 Connection-Mode Service 3
      2.1.1 Local Management 3
      2.1.2 Connection Establishment 7
      2.1.3 Data Transfer 10
      2.1.4 Connection Release 11
   2.2 Connectionless-Mode Service 12
      2.2.1 Local Management 12
      2.2.2 Data Transfer 12
   2.3 Advanced Topics 13
3. THE SOCKET INTERFACE 15
   3.1 Introduction 15
      3.1.1 `SOCK_STREAM`/TCP Socket Use 15
      3.1.2 `SOCK_DGRAM`/UDP Socket Use 19
   3.2 Advanced Topics 20
      3.2.1 Options 20
      3.2.2 Using `select` 21
4. REFERENCES 22

INTERACTIVE TCP/IP
Programmer's Supplement

1. INTRODUCTION

This document presents supplemental information about how to program two interfaces of the Transmission Control Protocol and Internet Protocol (TCP/IP) product under the INTERACTIVE UNIX* Operating System, enhanced by INTERACTIVE Systems Corporation. The Internet Protocol (IP) is not described here. Refer to *ip*(7) in the *INTERACTIVE TCP/IP Guide* for more information about the Internet Protocol. This supplement specifically describes the INTERACTIVE TCP/IP product, but it is generally applicable to the NP622 product as well.

1.1 The TCP/IP Interfaces

TCP/IP supports two programming interfaces: the USL Transport Layer Interface (also referred to as TLI or the Transport Interface) and the Berkeley Software Distribution (BSD) socket interface. The TLI is a library of routines and state transition rules that provides the basic data transfer service required by higher layer protocols, as supported by TCP/IP. For more information about TLI, refer to the *Network Programmer's Guide*. The socket interface is a program interface mechanism that provides endpoints for communication between processes. For more information about the socket interface, refer to Section 3i of the *UNIX Programmer's Reference Manual* [1].

Both of these interfaces provide similar access to the data transfer services of the underlying network protocols but differ in programming techniques. Because of their equivalence, the use of one interface rather than the other is largely a matter of preference, based on the user's programming experience or on the requirements of the environment in which an application is to run. Further comparison is not made here, but more details about each interface may be obtained from the reference manuals listed in the next section.

1.2 Prerequisites

For the Transport Interface, users must have a working knowledge of the networking support functions described in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*, Section 3N,

including all functions except `t_rcvrel`. In addition, users should be well acquainted with the corresponding version of the *Network Programmer's Guide*, which this guide is designed to supplement. In the *Network Programmer's Guide*, examples are based on a client-server paradigm; familiarity with this paradigm is assumed. For more information about this and other reference documentation, see the "Documentation Roadmap" in the *INTERACTIVE UNIX Operating System Guide* and section 4 of this document.

The TCP/IP socket interface is similar to the Release 4.3BSD socket mechanism, as documented in *socket*(2) of the 4.3BSD *UNIX Programmer's Reference Manual*. Section 3i provides documentation for the INTERACTIVE TCP/IP version of the 4.3BSD socket mechanism. Readers should have working knowledge of this material.

1.3 Overview of This Document

This document has four main sections:

1. INTRODUCTION
   This section introduces the interfaces, specifies the prerequisite knowledge necessary, lists reference documentation, and describes the organization of this guide.
2. THE TRANSPORT LAYER INTERFACE
   This section presents supplemental details about the Transport Layer Interface and provides examples of how to use this interface. It also provides an explanation of the two types of services available: connection mode and connectionless mode.
3. THE SOCKET INTERFACE
   This section describes how to program the socket interface. It provides instructions for using two types of sockets, SOCK_STREAM and SOCK_DGRAM, that correspond to the connection-mode and connectionless-mode services of the Transport Interface.
4. REFERENCES
   This section provides information on related documents.

2. THE TRANSPORT LAYER INTERFACE

The TCP/IP Transport Layer Interface supports two types of services: connection mode and connectionless mode. Connection-mode service reliably transfers data in proper sequence through a single, two-way data stream. This type of service is provided by the Transmission Control Protocol (TCP) module of TCP/IP, which runs on top of the Internet Protocol (IP) layer. Connectionless mode provides a message-passing mechanism in which a transport endpoint can independently address each message sent. In this mode, delivery of messages may not be reliable. In the TCP/IP product, the User Datagram Protocol (UDP) and IP modules are such unreliable services. UDP also runs on top of the IP layer.

2.1 Connection-Mode Service

The connection-mode service provided by the TCP module has four phases as described in Chapter 3 of the *Network Programmer's Guide*: local management, connection establishment, data transfer, and connection release. The following sections present supplemental details about these phases.

2.1.1 Local Management

This section supplements "Local Management" in Chapter 3 of the *Network Programmer's Guide*.

2.1.1.1 `t_open`. `t_open` establishes a TCP transport endpoint – a local channel to the transport provider. For example, a transport provider with connection-mode service could be opened by specifying `/dev/tcp` as the path to the transport provider (the first argument to `t_open`) and `O_RDWR` as the open flag (the second argument to `t_open`).

If a `t_info` structure is specified as the third argument of the `t_open` call, `t_open` sets the fields of the `t_info` structure with the following values, which characterize this protocol:

Field	*Value*	*Comment*
addr	8	
options	−1	Although the option size has no set limit, the number of options possible defines a practical limit.
tsdu	0	
etsdu	0	
connect	-1	
discon	-1	
servtype	T_COTS	

Otherwise, if these values are not needed by the user, `info` may be set to `NULL` so no values are returned. Refer to Chapter 3 of the *Network Programmer's Guide* for more information about these fields.

2.1.1.2 `t_bind`. `t_bind` is called to bind a transport provider stream (address) to a transport endpoint. The TCP provider uses the following structure in the `addr.buf` field of the *req* parameter:

```
struct sockaddr_in {
        short           sin_family;
        u_short         sin_port;
        struct in_addr  sin_addr;
        char            sin_zero[8];
};
```

This structure is defined in `<netinet/in.h>`. Only the first eight bytes of the structure are relevant to TCP. The `sin_family` field must be set to the value `AF_INET` as defined in `<sys/socket.h>`. The `sin_port` and `sin_addr` fields must be in network byte order, which on the Intel* 80386* microprocessor is the exact reverse of native byte order. (For information about library routines that manipulate byte order, see section 3.1.1.1 of this document.) The last member of the structure, `sin_zero`, is a filler that is not required by the Transport Interface.

If the programmer specifies the *req* parameter as `NULL` or the port and address values as zero, the TCP provider assigns an address to the transport endpoint. TCP assigns the port or the address value if either of these is unspecified.

If a `ret` field is provided, the TCP provider returns the address bound to the transport endpoint. The `addr` field of `ret` will

normally return a zero value unless a nonzero address value was provided in the `req` parameter. The reason for the zero value is that TCP defers choosing an endpoint address until a connection is established, at which point the address of the interface the connection will use becomes the bound address. If there was only one possible address, this would not be necessary, but many TCP/IP implementations, including INTERACTIVE TCP/IP, have at least two: the LAN interface and the loopback interface.

Some restrictions affect the port number to which an endpoint can bind; for example, only a privileged process can bind to a port number less than 1024. In general, only a server process must bind to a particular port number. Client processes and other uses commonly allow the system to assign the port.

2.1.1.3 `t_optmgmt`. Usually, local management of the TCP transport endpoint does not require setting any protocol options; however, the `t_optmgmt` function is available to change some of the options and operating parameters that TCP and IP use. Because these options are specific both to the protocols and to their implementations, setting options reduces portability.

For the INTERACTIVE TCP/IP product, the protocol options are classified in these categories:

- Socket: program-interface related
- TCP: TCP-specific
- UPD: UDP-specific
- IP: IP-related

The option values used by TCP are identical to those used by the socket interface. Socket options are defined in `<sys/socket.h>`. The TCP, UDP, and IP options are found in the files `<sys/tcp.h>`, `<sys/udp.h>`, and `<sys/ip.h>`, respectively. For information about these options, see *tcp*(7), *udp*(7), and *ip*(7) in the *INTERACTIVE TCP/IP Guide*.

Protocol options are specified as a sequence of structures in the `opt` field of the *req* parameter in the `t_optmgmt` call:

```
struct inetopt {
   short     len;            /* length of option (including lead-in) */
   ushort    name;           /* option identifier */
   ushort    level;          /* socket TCP, UDP, or IP, etc. */
   short     fill;           /* filler to align word */
   union inetoptval {
         short      sval;    /* shortint value */
         int        ival;    /* int value */
         long       lval;    /* long int value */
         caddr_t   *cval;    /* byte string value */
         unchar     vval[1]; /* arbitrary length data */
   }     value;
};
```

Boolean values are represented by a nonzero `ival` for true and by a zero `ival` for false. Variable-length data is an arbitrary number of bytes referenced by the `vval` field. The length of the data in the `vval` field is determined by the `len` field of the structure minus the size of the structure, excluding the size of the union. All current options use either the `ival` or the `vval` data forms.

For TCP, the most useful options are `SO_SNDBUF` and `SO_RCVBUF`, which change the size of the *send* and *receive* buffers, and the value of the TCP flow-control window. The option `TCP_NODELAY`, a Boolean, can modify the algorithm used to delay sending small packets in an attempt to improve performance.

2.1.1.4 Client Example. This example indicates how a client creates a transport endpoint before connection is established:

```
#include <tiuser.h>
#include <fcntl.h>
#include <stropts.h>
#include <sys/types.h>
#include <sys/bsdtypes.h>
#include <netinet/in.h>
#include <sys/socket.h>


main()
{
   int fd;

   /* open a TCP stream */
   if ((fd = t_open("/dev/tcp", O_RDWR, NULL))<0){
      t_error("t_open");
      return -1;
   }

   /* bind to whatever TCP assigns by default */
   if (t_bind(fd, NULL, NULL)<0){
      t_error("t_bind");
      t_close(fd);
      return -1;
   }
```

Refer to the *Network Programmer's Guide* for more information.

2.1.1.5 Server Example. This example indicates how a server creates a transport endpoint before connection is established:

```
#include <tiuser.h>
#include <fcntl.h>
#include <stropts.h>
#include <sys/types.h>
#include <sys/bsdtypes.h>
#include <netinet/in.h>
#include <sys/socket.h>

struct t_bind *bind;
struct t_call *call;
#define ADDRESS struct sockaddr_in *

main()
{
   int fd, newfd;

   /* open a TCP stream */
   if ((fd = t_open("/dev/tcp", O_RDWR, NULL))<0){
      t_error("t_open");
      /* some type of error recovery */
   }

   /* allocate data structure for a bind to specific address */
   /* then fill in the information required                  */
   if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR))
   == NULL){
      t_error("t_alloc");
      /* some type of error recovery */
   }
   bind->qlen = 5;
   bind->addr.len = ADDRESS_SIZE;
   ((ADDRESS)bind->addr.buf)->sin_port = LOCAL_PORT;
   ((ADDRESS)bind->addr.buf)->sin_addr.s_addr = LOCAL_ADDRESS;

   /* bind to the specified address */
   /* return value of "bind" is address that was assigned */
   if (t_bind(fd, bind, bind)<0){
      t_error("t_bind");
      exit(1);
   }
```

In this example, `ADDRESS_SIZE`, `LOCAL_PORT`, and `LOCAL_ADDRESS` represent appropriate values, which can be obtained as described in section 3.1.1.1.

2.1.2 Connection Establishment

This section supplements "Connection Establishment" in Chapter 3 of the *Network Programmer's Guide*.

TCP, a connection-oriented protocol, establishes transport endpoints that take one of two forms:

- Active: performs a `t_connect` operation.

- Passive: waits for a connection request through `t_listen` and acts upon the request with `t_accept`.

Active endpoints are typically used in network client applications, and passive endpoints are typically used in network server applications.

2.1.2.1 Client. A network client application uses an active endpoint. As noted in the *Network Programmer's Guide*, `t_connect` requires a *sndcall* argument that is a pointer to a `t_call` structure. For TCP, this `t_call` structure must have, at a minimum, a `T_ADDR` field. (In `t_call`, the `buf` field of the *addr* `netbuf` structure must provide the address of the remote TCP server in the form of a `sockaddr_in` structure.) The `sin_addr` field of the `sockaddr_in` structure must contain the network-byte-order representation of the remote system's internet address. The `sin_port` field of the structure must contain the network-byte-order representation of the remote server's well-known port address. The `sin` family field must contain `AF_INET`.

If the connection request is rejected or the remote system is unreachable, `t_connect` will fail and set `t_errno` to `TLOOK`. In this case, the `t_look` routine can be used to determine the event causing the failure. The `T_DISCONNECT` event indicates connection failure. To determine the exact cause of failure, the programmer can use the `t_rcvdis` function and examine the `reason` field. Possible reasons for failure include:

ETIMEDOUT	Connection not established before the time-out limit.
ECONNREFUSED	Connection refused by the remote system (which may not support the service or may have restricted access to the service).
ENETUNREACH	The network specified in the destination address cannot be reached from this system.

Additional events that may set `t_errno` to `TLOOK` are described in the *Network Programmer's Guide*. Exceptions are `T_ORDREL` and `T_UDERR`, which are not supported in INTERACTIVE TCP/IP.

2.1.2.2 Client Connection Example. In the following example, the client requests a connection:

```
/* allocate data structure for connect address */
/* then fill in the necessary information      */
if ((call = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR))
== NULL){
   t_error("t_alloc");
   t_close(fd);
   return -1;
}
call->addr.len = ADDRESS_SIZE;
((ADDRESS)call->addr.buf)->sin_family = AF_INET;
((ADDRESS)call->addr.buf)->sin_port = REMOTE_PORT;
((ADDRESS)call->addr.buf)->sin_addr.s_addr = REMOTE_ADDRESS;

/* attempt the connection */
if (t_connect(fd, call, NULL) < 0){
   t_error("connect");
   t_close(fd);
   return -1;
}
return fd;
}
```

In this example, `ADDRESS_SIZE`, `REMOTE_PORT`, and `REMOTE_ADDRESS` represent appropriate values, which can be obtained as described in section 3.1.1.1. `ADDRESS_SIZE` is the same size as `sockaddr_in`, which is described in section 2.1.1.2

2.1.2.3 Server. Typically, a network server application uses a passive endpoint, accepting on a new stream any incoming connect requests received by the `t_listen` function. After a new stream is opened by the `t_accept` function, the server can accept another connect request.

The server must allocate a `t_call` structure including, at a minimum, a `T_ADDR` field. The `t_listen` function normally waits for an event and returns with success on receipt of a connect request from another endpoint. The `t_listen` function also returns the `t_call` structure which specifies the address of the requesting system. The `sockaddr_in` structure contained in the address buffer of the `t_call` structure will provide both the port and the internet addresses of the remote system. When a connect request is received, the application either accepts the connection with a `t_accept` or rejects it with a `t_snddis` call.

2.1.2.4 Server Connection Example. In the following example, a server calls `t_listen` to listen for an incoming connection request, and `t_accept` to accept a request for connection:

```
/* allocate data structure for the accept address */
if ((call = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL){
   t_error("t_alloc");
   /* some type of error recovery */
}

/* listen for a connection */
/* "call" will have the requesting address */
if (t_listen(fd, call)<0){
   t_error("listen");
   /* some type of error recovery */
}

/* open new TCP stream to associate the connection with */
if ((newfd = t_open("/dev/tcp", O_RDWR, NULL))<0){
   t_error("t_open");
   exit(1);
}

/* bind it to any address */
if (t_bind(newfd, NULL, NULL)<0){
   t_error("t_bind");
   t_close(newfd);
   /* probably a fatal error */
   exit(1);
}

/* accept the requested connection on the new stream */
/* descriptor if possible */
if (t_accept(fd, newfd, call) < 0){
   t_error("accept");
   if (t_errno == TLOOK){
      /* there is some possibility of a disconnect */
      /* request to this address */
      if (t_rcvdis(fd, NULL) < 0){
         t_error("rcvdis");
         exit(1);  /* fatal error */
      }
   }
   t_close(newfd);
   newfd = -1;
}

/* newfd is either an error indication or the accepted connection */
```

A connection can be accepted on the same transport endpoint that receives it. If this occurs, the stream so opened must disconnect before the endpoint can accept additional connection requests.

2.1.3 Data Transfer

This section supplements "Data Transfer" in Chapter 3 of the *Network Programmer's Guide.*

After a connection is established, data can be transferred by using the `t_snd` and `t_rcv` calls, which function as the `read` and `write` system calls, respectively. This example shows how normal data is sent:

```
if (t_snd(fd, buff, len, 0)<0)
   t_error("t_snd");
```

The TCP transport provider does not support transfer of expedited data, which has priority over other data. Instead, TCP supports *urgent data*. Urgent data is sent as part of the data stream, along with any other data queued but not yet sent. Urgent data has an indication of its end but not of its quantity (unlike expedited data). Accepted usage assumes one byte of data. To transfer urgent data, the programmer should use the `T_EXPEDITED` flag to set the TCP `URGENT` pointer. If multiple `URGENT` pointers are sent close together, they can be lost or overwritten by a receiving TCP.

TCP supports data transfer in byte stream mode, but not in Transport Service Data Units (TSDUs). Consequently, the `T_MORE` flag is ignored on send and is always zero on receive.

2.1.4 Connection Release

This section supplements "Connection Release" in Chapter 3 of the *Network Programmer's Guide*.

A connection can be terminated by either user at any point during data transfer. Currently, TCP provides only an abortive disconnect, initiated by the `t_snddis` call. Data may be sent with the disconnect request, but its delivery is not guaranteed.

☛ Note that the INTERACTIVE TCP/IP product will guarantee the delivery of data accompanying the disconnect request when connected to a host running the INTERACTIVE TCP/IP product. When connected to a host with a different TCP/IP implementation, however, the delivery of data accompanying a disconnect request may not be supported. It is safe to assume, in general, that the delivery of data sent with the disconnect request is not guaranteed.

If the remote endpoint requests the disconnection, the local endpoint receives an error that causes `TLOOK` to notify the application of the event. The `TLOOK` is a `T_DISCONNECT` event. `t_rcvdis` can be used to return the disconnect request and the associated reason. Commonly, the reason is zero, which indicates a normal disconnect.

2.2 Connectionless-Mode Service

Connectionless-mode service is provided by the User Datagram Protocol (UDP) module of the TCP/IP product. This mode has two phases: local management and data transfer.

2.2.1 Local Management

This section supplements "Local Management" in Chapter 4 of the *Network Programmer's Guide*.

2.2.1.1 `t_open`. `t_open` is called to create a UDP transport endpoint. Specify the file `/dev/udp` as the STREAMS clone device node and `O_RDWR` as the open flag.

If the third parameter of `t_open`, the `info` field, is supplied, the `t_info` structure has the following characteristic values:

Field	*Value*	*Comment*
`addr`	8	
`options`	-1	Although no limit is set on the option size, the number of options possible defines a practical limit.
`tsdu`	0	
`etsdu`	0	
`connect`	-1	
`discon`	-1	
`servtype`	T_CLTS	

2.2.1.2 `t_bind`. The transport endpoint must be bound to an address using the same conventions as for TCP, described in section 2.1.1.2.

2.2.2 Data Transfer

This section supplements "Data Transfer" in Chapter 4 of the *Network Programmer's Guide.*

UDP data transfer uses the `t_sndudata` and the `t_rcvudata` calls. A `T_UNITDATA` structure must be allocated. Each datagram sent must specify the destination address of the data and reference the data. Operational values may be specified in the `options` field.

One important option is the Boolean `SO_BROADCAST`, which is required to send a message to the broadcast address. IP options

may also be used with UDP datagrams. For more information about protocol options, see section 2.1.1.3 in this document and *udp*(7) in the *INTERACTIVE TCP/IP Guide*.

2.3 Advanced Topics

The Transport Interface has additional features that are useful in some applications. Of particular interest are those that support asynchronous processing. All of the examples in Chapter 6 of the *Network Programmer's Guide* apply to the TCP/IP product. In addition, this section describes how to prevent blocking for asynchronous processing by using the the STREAMS `poll` system call.

The `poll` system call can be used to determine which stream file descriptors have data available for receiving and which are not blocked from sending. `poll`, a relatively easy-to-use asynchronous-event interface, allows the programmer to multiplex input/output over a set of stream file descriptors without blocking on any stream. Refer to the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

`poll` takes an array whose elements are `pollfd` structures of the following form:

```
struct pollfd {
        int     fd;       /* file descriptor */
        short   events;   /* events of interest on fd */
        short   revents;  /* events that occurred on fd */
};
```

For a specified file descriptor, `fd`, the above structure is used to define the events of interest.

Two strategies enable an application to perform tasks rather than to remain in the `poll` call. One is to specify the `poll` time-out value as zero, which immediately returns the system call instead of allowing it to wait for an event that might otherwise block. Alternatively, the programmer can use the `I_SETSIG ioctl` command with a signal handler set to catch `SIGPOLL` signals. The `SIGPOLL` signal is sent only when a condition specified in the `I_SETSIG` command is satisfied. Usually, this condition is a message arriving at the stream head.

In addition to using `poll`, the programmer can prevent blocking by using the `O_NDELAY` flag on the `t_open` call when creating a file descriptor, so that the system calls will return errors of `TNODATA` when a receive would otherwise block, or `TFLOW` when a send would otherwise block.

3. THE SOCKET INTERFACE

3.1 Introduction

A *socket* is an abstraction of a communications endpoint that functions as a program interface. The TCP/IP socket interface is identical to the Release 4.3BSD socket mechanism, except as noted in this guide. For information about the 4.3BSD-compatible interface and the *–linet* library, refer to Section 3I of the manual entries included in the *INTERACTIVE TCP/IP Guide*.

The TCP/IP socket interface requires these `include` files:

File	*Description*
`<sys/socket.h>`	Socket-addressing information
`<netinet/in.h>`	Socket-addressing information
`<sys/bsdtypes.h>`	4.3BSD data types used to define various data structures
`<net/errno.h>`	Network-specific errno values

The first two files in this list are familiar to programmers experienced with the 4.3BSD socket. The other files listed are additional files required for the present implementation of TCP/IP.

A socket is created with the `socket` call. The socket descriptor is created in a specified communications domain conforming to a particular protocol model. This TCP/IP implementation supports the `AF_INET` (internet protocol family) domain with the protocol models `SOCK_STREAM` and `SOCK_DGRAM`.

`SOCK_STREAM` is a reliable, connection-based, byte stream model that does not preserve any records of `write` boundaries. In contrast, `SOCK_DGRAM` is an unreliable datagram, or message, model. In function and use, `SOCK_STREAM` corresponds to the TCP interface, and `SOCK_DGRAM`, to the UDP interface.

A third model, `SOCK_RAW`, provides direct access to the IP layer. Because of the infrequency of its use and its similarity to `SOCK_DGRAM`, `SOCK_RAW` is not documented here.

3.1.1 `SOCK_STREAM`*/TCP Socket Use*

`SOCK_STREAM` sockets can exist in two forms: active, in which the socket actively establishes connections, and passive, in which the socket passively waits for an incoming connection request. In both

forms, a connection must be established before data can be transferred.

3.1.1.1 Creating a TCP Socket. A TCP socket is created in this way:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

After the socket is created, it should be bound to an address. Addresses are defined in the `sockaddr_in` structure, documented in <netinet/in.h>, which has this definition:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};

struct in_addr {
        unsigned long s_addr;
};
```

In this structure, the `sin_family` field is always `AF_INET`. The `sin_addr` field is an internet address in network byte order. The `sin_port` field is a TCP port address in network byte order. The value of the `sin_zero` field is ignored.

☛ Note that the `sockaddr_in` structure is used to pass socket address information to the socket interface routines. The programmer who is not familiar with these routines, which are described in Section 3I of the manual entries, will notice that another structure, the `sockaddr` structure, is expected. Although these are two separately defined structures, the data they contain are the same.

Network byte order specifies that byte zero is the most significant in a word. On the Intel 80386 microprocessor, network byte order is the exact reverse of native byte order. Although the examples in this guide use a symbolic constant to represent the host and port addresses (assumed in network byte order), these values are usually determined dynamically.

The *–linet* library provides several functions for manipulating byte order, as well as for determining host and port addresses from their symbolic names. For more information about these functions, consult the following manual entries in the *INTERACTIVE TCP/IP Guide*.

Function	*Manual Entry*	*Description*
ntohs	*byteorder*(3I)	Convert short int from network to host order.
ntohl	*byteorder*(3I)	Convert long int from network to host order.
htons	*byteorder*(3I)	Convert short int from host to network order.
htonl	*byteorder*(3I)	Convert long int from host to network order.
gethostbyname	*gethostent*(3I)	Get host address by name.
getservbyname	*getservent*(3I)	Get service (port) by name.

Section 3I of the manual entries in the *INTERACTIVE TCP/IP Guide* describes additional utility functions of interest.

To bind an address to the socket, a `sockaddr_in` structure is set up with appropriate values, and a call is made to `bind`:

```
struct sockaddr_in myaddr;
   .
   .
   .
myaddr.sin_family = AF_INET;
myaddr.sin_port = LOCAL_PORT;
myaddr.sin_addr.s_addr = 0;
if (bind(s, &myaddr, sizeof myaddr)<0){
   perror("bind");
   exit(1);
}
```

In this example, `s` is a descriptor returned by a *socket*(3I) call and `LOCAL_PORT` represents an appropriate value. If the value of `LOCAL_PORT` is zero, `bind` will select an unused port with a value of 1024 or greater. Typically, only a server must bind to a specific port address. If either the port or internet address is nonzero, it must be valid for the system on which the bind is done. Refer to section 2.1.1.2 in this document and to the *socket*(3I) manual entry in the *INTERACTIVE TCP/IP Guide* for more information.

3.1.1.2 Connection Establishment. The client uses an active socket to establish a connection through the `connect` call. The remote address must be specified fully (that is, the *port* and *addr* values must be nonzero). The following example illustrates a client establishing connection:

```
struct sockaddr_in remote;
   .
   .
   .
remote.sin_family = AF_INET;
remote.sin_port = REMOTE_PORT;
remote.sin_addr.s_addr = REMOTE_ADDRESS;
if (connect(s, &remote, sizeof(remote))<0){
   perror("connect");
   exit(1);
}
```

In this example, `s` is a descriptor returned by a *socket*(3I) call and `REMOTE_PORT` and `REMOTE_ADDRESS` represent appropriate values, which can be obtained by making calls such as the `gethostbyname` and `getservbyname` calls, as described in section 3.1.1.1. TLI does not provide a standard mechanism for determining these values.

The server uses a passive socket to listen for a client's request for a connection. A passive socket must wait for a connection request from a remote system. To do this, the socket first enters the passive mode with a `listen` call, then waits in an `accept` call. When a connect request arrives, the `accept` call accepts the connection and returns a new socket descriptor that is connected to the remote system. `accept` also returns the requestor's address. The original socket descriptor can then listen for a new connection request.

This example demonstrates the server listening for a request and then waiting in the `accept` call:

```
struct sockaddr_in remote;
int addrlen;
int newsocket;
   .
   .
   .
listen(s, 5);
remote.sin_family = AF_INET;
if ((newsocket = accept(s, &remote, &addrlen))<0){
   perror("accept");
   exit(1);
}
```

3.1.1.3 Data Transfer. After a connection is established, the programmer simply uses the `read` and `write` system calls to exchange data. TCP does not preserve record boundaries, so it is unwise to assume, for example, that when an application does 512-byte `writes`, the remote system receives data 512 bytes at a time. The remote system may receive the data in smaller or larger quantities, depending upon timing and various network

parameters. (Note that the size of the buffer used in the `sendto` call is completely arbitrary.)

3.1.2 SOCK_DGRAM/UDP Socket Use

`SOCK_DGRAM` sockets differ from `SOCK_STREAM` sockets in that they lack active and passive forms. In general, `SOCK_DGRAM` sockets do not establish connections; therefore, each outgoing message must have an associated destination address provided, and each incoming message, an associated source address. The `connect` call can be used to associate a destination address with a `SOCK_DGRAM` socket permanently, relieving the application of the need to provide an address with each message. Permanent association of an address enables use of the `write` call on a `SOCK_DGRAM` socket.

3.1.2.1 Creating a UDP Socket. A UDP socket is created in this way:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

After the socket is created, it should be bound to an address. Addresses are specified in a struct `sockaddr_in`, as described for TCP in section 2.1.1.2.

3.1.2.2 Connection Establishment and Data Transfer. The UDP does not establish connections. Each datagram sent has an explicit destination address associated with it, and each datagram received has a source address associated with it.

To send a datagram, the programmer commonly uses the `sendto` call:

```
struct sockaddr_in remote;
char buffer[512];
  .
  .
  .
remote.sin_family = AF_INET;
remote.sin_port = REMOTE_PORT;
remote.sin_addr.s_addr = REMOTE_ADDRESS;
cc = sendto(s, buffer, sizeof(buffer), 0, &remote, sizeof(remote));
```

In this example, `REMOTE_PORT` and `REMOTE_ADDRESS` represent appropriate values, which can be obtained as described in section 3.1.1.1.

Typically, a datagram is received by using the `recvfrom` call:

```
struct sockaddr_in remote;
char buffer[512];
int addrlen;
  .
  .
  .
addrlen = sizeof(remote);
cc = recvfrom(s, buffer, sizeof(buffer), 0, &remote, &addrlen);
```

In this example, *remote* will contain the source address of the datagram on return from `rcvfrom`. Note that the address of the *addrlen* variable is passed to `recvfrom`. The actual length of the address is returned; the original value is the size of the buffer that stores the address. The value `cc` returned by `recvfrom` is the amount of data received.

3.2 Advanced Topics

The socket interface provides additional features for the experienced programmer. Some of these are described here, and additional details are found in the relevant manual entries.

3.2.1 Options

In the socket model, options can be set on a socket descriptor. An option may affect the socket interface or a protocol below the socket interface. All options are set by the `setsockopt` call, which has this form:

```
int buflen;
  .
  .
  .
buflen = 8192;
setsockopt(s, SOL_SOCKET, SO_SNDBUF, &buflen, sizeof buflen);
```

The example adjusts the buffersize allocated for output buffers to 8192 bytes (the default is 4096). A possible use for this is in high-volume connections.

For TCP, the `TCP_NODELAY` option may be useful, especially in applications that send data but get no echo. Refer to *tcp*(7) in the *INTERACTIVE TCP/IP Guide* for more information.

Current option values are obtained with the `getsockopt` call. The option values used by the socket interface are identical to those used by TLI. Socket options are defined in `<sys/socket.h>`. For more information on the `getsockopt` and `setsockopt` calls and for a list of the socket level options, refer to *getsockopt*(3I) in the *INTERACTIVE TCP/IP Guide*.

3.2.2 *Using* `select`

The `select` call is used to determine whether a socket can be read or written without blocking. `select` takes bitmasks that specify the file descriptor(s) of interest. Three bitmasks can be used to check input, output, and urgent conditions. A time-out value is also specified. Normally, `select` blocks until one of the conditions being checked comes true. Blocking occurs when the time-out pointer is `NULL`. If the time-out pointer is non-null, the call blocks for the time specified in the `timeval` structure. A time-out value of zero causes `select` to return immediately.

In the INTERACTIVE UNIX System, `select` essentially provides the same functions as the `poll` system call but uses a different syntax. `select` has the same restrictions as `poll`, but under the INTERACTIVE UNIX System, `poll` has been extended to support non-STREAMS file descriptors. In particular, `pipes`, `pty`, and `tty` devices are supported.

4. REFERENCES

[1] *UNIX Programmer's Reference Manual*, 4.3 Berkeley Software Distribution Virtual VAX-1 Version. Berkeley: University of California, 1986.

In addition, the following documents are recommended:

Defense Communications Agency. *DDN Protocol Handbook, Vol I: DOD Military Standard Protocols*. "Military Standard Internet Protocol," U.S. Department of Defense MIL-STD-1777. Menlo Park, CA: DDN Network Information Center, 1985.

Defense Communications Agency. *DDN Protocol Handbook, Vol I: DOD Military Standard Protocols*. "Military Standard Transmission Control Protocol," U.S. Department of Defense MIL-STD-1778. Menlo Park, CA: DDN Network Information Center, 1985.

Defense Communications Agency. *DDN Protocol Handbook, Vol II: DARPA Internet Protocols*. "Internet Protocol – DARPA Internet Program Protocol Specification" by J. Postel, RFC 791. Menlo Park, CA: DDN Network Information Center, 1985.

Defense Communications Agency. *DDN Protocol Handbook, Vol II: DARPA Internet Protocols*. "Transmission Control Protocol – DARPA Internet Program Protocol Specification" by J. Postel, RFC 793. Menlo Park, CA: DDN Network Information Center, 1985.

INDEX

abortive release, TCP TLI 11
accept 18
addresses of remote system 4
addresses of remote system, SOCK_STREAM 17
addresses of remote system, TCP 9
addresses of remote system, TCP TLI 8
addresses of remote systems, SOCK_STREAM 16
asynchronous processing 13
binding SOCK_DGRAM socket 19
binding SOCK_STREAM socket 16, 17
binding transport endpoint, TCP 4, 5
blocking 21
byte order, network 4, 8, 16
byte-order manipulation 16
clone open, UDP TLI 12
connect 17
connection establishment, SOCK_STREAM 15, 17
connection establishment, TCP client 8
connection establishment, TCP TLI 7
connection failure, TCP TLI 8
connection mode 3
connection mode, description of 3
connection release, TCP TLI 11
connection request acceptance, TCP server 9
connection request, SOCK_STREAM server 18
connection request, TCP TLI client 9
connectionless mode 12
connectionless mode, description of 3
data transfer, expedited TCP TLI 11
data transfer, in TDSUs 11
data transfer, SOCK_DGRAM datagrams 19
data transfer, TCP TLI 10, 11
data transfer through SOCK_STREAM socket 18
data transfer, UDP TLI datagrams 12
data transfer, urgent TCP TLI 11
datagram addressing, SOCK_DGRAM 19
datagram addressing, UDP TLI 12
datagram model, socket 15
datagram, SOCK_DGRAM transfer of 19
expedited data 11
getsockopt 20
Internet Protocol 3
IP layer 3, 15
I_SETSIG 13
listen 18
local management, TCP TLI 3
local management, UDP Transport Interface 12
netbuf 8
network byte order 4, 8, 16
options for UDP datagram transfer 12
poll 13
pollfd 13
port, binding to 5
protocol options 5
read 18
recvfrom 19
remote system addresses 4
remote system addresses, SOCK_STREAM 16
remote system addresses, TCP TLI 8, 9
select 21
sendto 19
setsockopt 20
SO_BROADCAST 12
sockaddr_in, SOCK_DGRAM use of 19
sockaddr_in, SOCK_STREAM use of 16, 17
sockaddr_in, TCP TLI use of 4, 8, 9
SOCK_DGRAM socket creation 19
SOCK_DGRAM socket model 15, 19
socket, active SOCK_STREAM 17
socket creation 15
socket creation, SOCK_DGRAM 19
socket creation, SOCK_STREAM 16
socket forms, SOCK_STREAM 15
socket interface, definition of 1
socket interface, description of 15
socket options 5, 20
socket, passive SOCK_STREAM 18
socket, SOCK_DGRAM model 15, 19
socket, SOCK_RAW model 15
socket, SOCK_STREAM model 15
SOCK_RAW socket model 15
SOCK_STREAM socket creation 16
SOCK_STREAM socket model 15
SO_RCVBUF 6
SO_SNDBUF 6
t_bind, TCP TLI use of 4
t_bind, UPD use of 12
t_call, TCP TLI use of 8, 9
t_connect 8
t_connect, SOCK_DGRAM use of 19
t_connect, TCP TLI use of 8
TCP module 3
TCP_NODELAY 6, 20
T_DISCONNECT 8
t_errno 8
timeval 21
t_info, TCP TLI use of 3
t_info, UDP TLI use of 12
t_listen, TCP TLI use of 9
TLOOK 8
t_open, TCP TLI use of 3
t_open, UDP Transport Interface use of 12
t_open, use to prevent blocking 14
t_optmgmt 5
Transmission Control Protocol' 3
transport endpoint, active TCP TLI 7
transport endpoint, binding a TCP TLI 4
transport endpoint, creating a UDP TLI 12
transport endpoint creation, by TCP TLI client 6
transport endpoint creation, by TCP TLI server 7
transport endpoint, passive TCP TLI 9
transport endpoint, TCP TLI 10
transport endpoints, forms of TCP TLI 7
Transport Layer Interface 3
Transport Layer Interface, definition of 1
Transport Service Data Units 11

t_rcv 10
t_rcvdis 8
t_rcvudata 12
t_recvdis 11
t_snd 10
t_snddis 11
t_sndudata 12
T_UNITDATA 12
UDP module 3, 12
urgent data 11
User Datagram Protocol 3, 12
write 18, 19

INTERACTIVE

DC00213-2Z