X/Open CAE Specification

Window Management (X11R5):
File Formats and Application Conventions

X/Open Company Ltd.

X/Open CAE Specification

Window Management (X11R5): File Formats and Application Conventions

ISBN: ISBN 1-85912-090-3

X/Open Document Number: C510

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

List of Figures

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to info-server@xopen.co.uk with the following in the Subject line:

```
request corrigenda; topic index
```

This will return the index of publications for which Corrigenda exist.

This Document

This document is a CAE specification (see above).

It contains four MIT **X Window** specifications as follows.

- Inter-Client Communications Conventions Manual (ICCCM)

  This X/Open specification is based on the following document:

  Inter-Client Communications Conventions Manual (ICCCM), Version 1.1
  MIT X Consortium
  by David Rosenthal, Sun Microsystems, Inc.

  Material for the chapter on Device Color Characterization was provided by Keith Packard.

  This specification refers to the X Protocol described in the **X Window System Protocol** specification.

- X Logical Font Description (XLFD)

  This X/Open interface definition is based on the following document:

  X Logical Font Description Conventions, Version 1.4
  MIT X Consortium

- Compound Text

  This X/Open interface definition is based on the following document:

  Compound Text Encoding, Version 1.1
  MIT X Consortium
  by Robert W.Scheifler, Laboratory for Computer Science,
  Massachusetts Institute for Technology

- Bitmap Distribution Format (BDF)

  This X/Open interface definition is based on the following document:

  Bitmap Distribution Format, Version 2.1
  MIT X Consortioum Standard

Structure

The source documents for this publication have undergone revision since X11R4, and this is reflected in the new or rearranged chapters of this publication.

- **Chapter 1** gives an overview of the X Window System, and is common to all four of the X/Open Window Management (X11R5) specifications.
- **Chapters 2 to 8** include the MIT ICCCM standard.
- **Chapters 9 to 12** are the MIT Logical font Description (XLFD) specification.
- **Chapter 13** is the MIT Compound Text specification.
- **Chapter 14** is the MIT Bitmap Distribution Format (BDF) specification.

X/Open Window Management Document Set

This specification is one of four specifications in the X/Open Window Management (X11R5) document set. The full set comprises:

- X Window System Protocol
- Xlib - C Language Binding
- X Toolkit Intrinsics
- File Formats and Application Conventions.

These X11R5 specifications are available as a 4-volume set (Document Number T410).

The following table shows the structure and organisation of material in this document set in terms of the MIT documentation of the X Window System, on which the X/Open document set is based.

In each document, Chapter 1 is an X/Open overview of the X Window System, which is not in the MIT documentation.

X/Open Document	Subject	MIT Document
X Window System Protocol	Description and definition of the X Protocol	X Window System Protocol
Xlib - C Language Binding Chapters 2-17 and Appendices A-D	Description of Xlib functions and their use	Xlib - C Language X Interface
Chapter 18	X/Open additional requirements	None
X Toolkit Intrinsics	Description of X Toolkit functions and their use	X Toolkit Intrinsics
File Formats and Application Conventions	Various formats and conventions for application cooperation and communication	Inter-Client Communication Conventions Manual (ICCCM), Version 1.1 X Logical Font Description (XLFD), Version 1.4 Compound Text, Version 1.1 Bitmap Distribution Format (BDF) 2.1

The X Window Management (X11 Release 5) System is required by the X/Open Common Desktop Environment (XCDE), which defines a common graphical user interface environment.

The other specifications in the XCDE family are:

- X/Open Common Desktop Environment (XCDE) 2-volume set comprising:
  - — Definitions and Infrastructure
  - — Services and Applications
- Motif Toolkit API (electronic publication)
- Calendaring and Scheduling API (XCS).

Trade Marks

UNIX® is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open® is a registered trade mark, and the ‘‘X’’ device is a trade mark, of X/Open Company Limited.

X Window System™ is a trade mark of the Massachusetts Institute of Technology.

Acknowledgements

X/Open would like to acknowledge the use of parts of the following documents: **Inter-Client Communication Conventions Manual (ICCCM)**, Version 1.1, Mit X Consortium; **X Logical Font Description Conventions**, Version 1.4, MIT X Consortium; **Compound Text Encoding**, Version 1.1, MIT X Consortium; and **Bitmap Distribution Format**, Version 2.1, MIT X Consortium.

Referenced Documents

The following documents are referenced in this specification:

X11R5 X Protocol
: X/Open CAE Specification, May 1995, Window Management (X11R5): X Window System Protocol (ISBN: 1-85912-087-3, C507).

X11R5 Xlib
: X/Open CAE Specification, May 1995, Window Management (X11R5): X Lib - C Language Binding (ISBN: 1-85912-088-1, C508).

X11R5 X Toolkit
: X/Open CAE Specification, May 1995, Window Management (X11R5): X Toolkit Intrinsics (ISBN: 1-85912-089-X, C509).

ISO 2022
: ISO 2022: 1986 Information Processing — ISO 7-bit and 8-bit Coded Character Sets — Coded Extension Techniques.

ISO 8859-1
: ISO 8859-1: 1987, Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.

ISO/IEC 6429
: ISO/IEC 6429: 1992, Information Technology — Control Functions for Coded Character Sets.

Chapter 1

Overview of the X Window System

1.1 Introduction

The X Window System is a network-transparent windowing system developed under the auspices of Project Athena at the Massachusetts Institute of Technology. The X Window System is implemented as a client-server model.‘ The window system functionality is provided by a display server, which is resident on a machine which has one or more monochrome or color raster displays attached. Client applications which require window system services attach to a server, and subsequently communicate with it, via an Inter-Process Communications connection. This uses a standard and extensible asynchronous protocol to communicate window system protocol requests to the server.

A client may, but not necessarily, run on the same machine as the X Server it is connected to. Applications may reside on hosts remotely connected to the system which hosts the display server by some kind of local or wide-area networking technology. This is dependent upon the level of functionality provided by the particular networking environment in which particular server and client implementations operate.

An X Window System server supports one or more physical, monochrome or color, raster streens, which display a logical hierarchy of (possibly) overlapping rectangular areas known as ‘‘windows’’. Also associated with the server is a number of input devices. Normally these include a keyboard and some form of pointing device, such as a mouse or digitising tablet.

At the top, or root, of the logical window hierarchy, is the ‘‘root window’’ which completely covers the physical screen with which the hierarchy is associated. In the normal course of operation, each ‘‘root window’’ will be partially, or completely, covered by ‘‘child windows’’ created by clients. Due to the organisation of the window hierarchy, an application program may create a tree of arbitrary depth on each screen. The X Window System Protocol provides applications with the functionality to create and manipulate windows and their associated attributes. The X Window System also provides the ability to associate arbitrary data with a window, access fonts and colors, perform general graphical output, and obtain input from the available devices, using a canonical, programmatic interface, which embodies a high degree of device independence.

A client that converses with the server using the X Window System protocol may operate ‘‘correctly’’ in isolation, but might not coexist properly with other clients sharing the same server. The **ICCCM** specification is a set of conventions to allow clients to cooperate in the areas of selections, cut buffers, window management, session management and resources.

1.2 X Window System Overview

The X Window System architecture is divided into two distinct parts (Figure 1-1):

display servers Provide display capabilities and keep track of user input.

clients Application programs that perform specific tasks.

This separation allows the clients and servers either to work together on the same system, or across a network. Regardless of where the clients are running, all user input and displayed output will occur on the workstation server. Communication is accomplished (in a network transparent fashion) using the X Protocol.

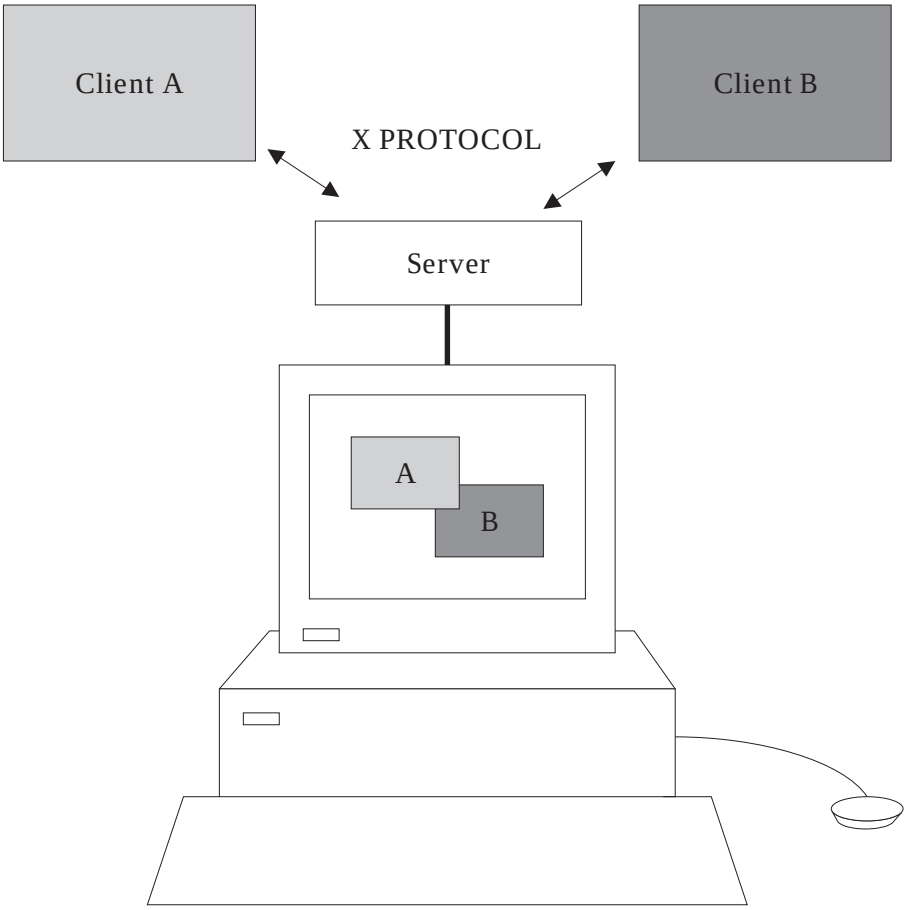


Figure 1-1 X Window System Overview

1.2.1 X Platform Abstraction Layers

The X Window System consists of several distinct parts. Figure 1-2 shows them as layers.

- The *X Protocol* defines the format and sequencing of byte streams and semantics (messages) passed between X Clients and the X Server.
- *Xlib* specifies the function call interface to build the messages defined by the X Protocol.
- The *Xt Intrinsics* provide the basic constructs to support the creation and use of user interface objects (widgets).

- The *Widgets* provide a set of user interface features (such as menus and pushbuttons) and allow applications to manipulate these features using object-oriented programming techniques.

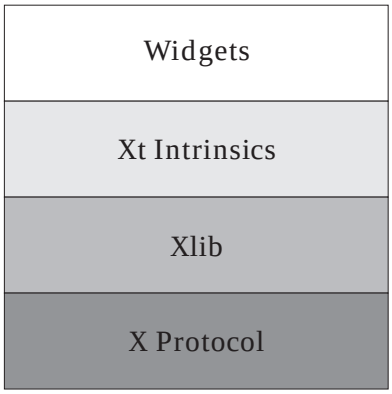


Figure 1-2 X Platform Abstraction Layers

1.2.2 User Interface Platform

From the programmer's perspective, the X Window System provides a *User Interface Platform* with multiple interfaces (Figure 1-3). Applications can be developed using any or all of these interfaces, depending on the requirements of the developer. It is important to note here that the lowest-level interface is Xlib – the X Protocol does not provide a practical programming interface. Therefore, all interaction with the X Protocol is handled by Xlib calls. It is not necessary to program directly using Xlib to create an X Window System application. Therefore, the interface boundaries should be viewed as transparent from a programmer's perspective (the programmer may use any or all of them to achieve the desired results in the program).

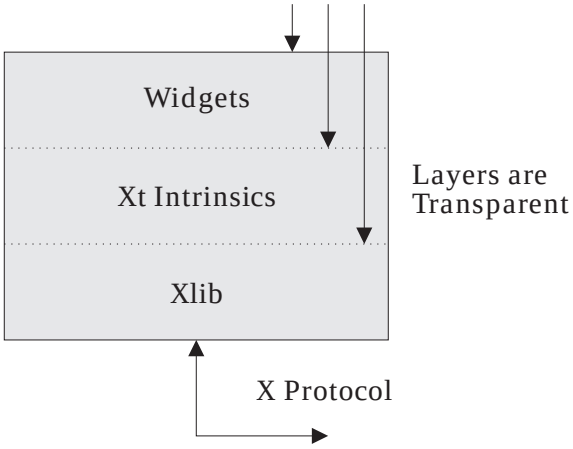


Figure 1-3 User Interface Platform

1.2.3 A Single X Application

The User Interface platform provides all the services necessary to manage the user interface aspects of the application. *Application functionality* is that part of the application which is independent of any user interface function, but it is the application that knows what it wants to accomplish through the user interface. The translation of the application's user interface needs into user interface actions or displays is achieved through a form of binding.

This binding can be an integral part of the application, indistinguishable from the application functionality, or it can be a separate module created by a development tool or language and stored in a separate library or binary module. The separation of *application functionality* from *user interface functionality* (in so far as it is possible) helps to provide application portability and ease of maintenance.

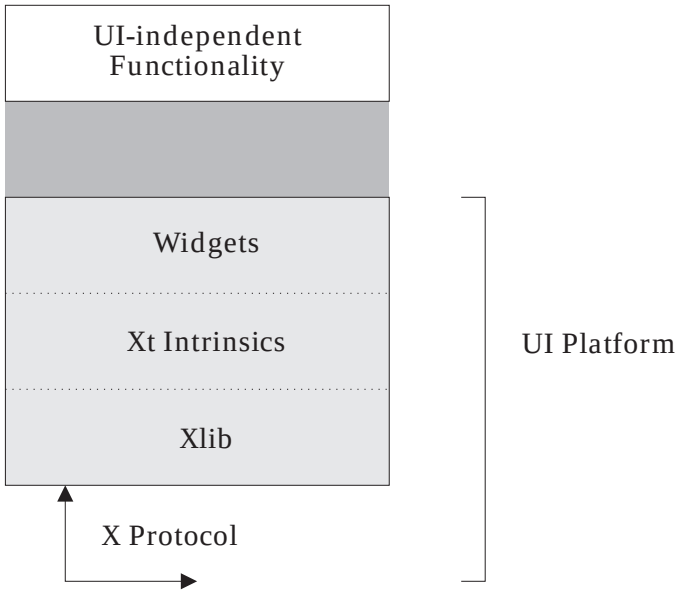


Figure 1-4 A Single X Application

1.2.4 X Application Relationships

The X Window System environment usually consists of several *client applications*, all communicating with an *X Server* at the same time using the *X Protocol* (Figure 1-5). Some of these clients have special roles within the environment, such as window and session managers. In order for all of these applications to work together *cooperatively*, *Inter-Client Communications Conventions* have been established. These ensure that client applications will cooperate in their use of the server and can also interact directly with each other.

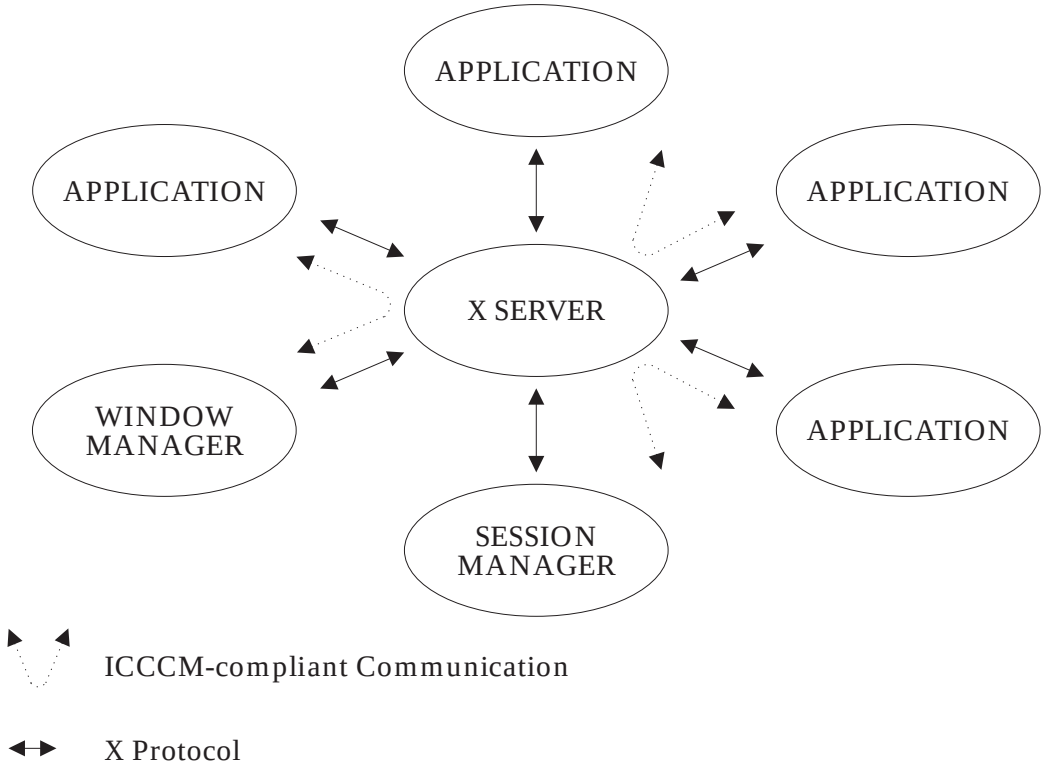


Figure 1-5 X Application Relationships

Window Management (X11R5)

Part 1:

Inter-Client Communications Conventions Manual (ICCCM)

X/Open Company Ltd.

Chapter 2

Introduction to ICCCM

2.1 Status

The conventions defined by this specification are semantically correct, and complete within those areas currently addressed.

All the facilities specified in this document are mandatory.

There are no parts of this specification which are optional.

Internationalisation

The X Window System is 8-bit transparent. Any 8-bit or 16-bit codeset may be used in the font and text calls. In addition, 8-bit codesets may be used in all strings including filenames, atom names and color names.

The ISO Latin-1 character set (ISO 8859-1) is used in generating atom names in this specification. Certain parts of this specification describe specific protocols for ISO Latin-1 strings.

2.2 Overview

It was an explicit design goal of the X Window System Protocol, 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** specification, 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 that govern inter-client communications in the following areas:

- selection mechanism
- cut buffers
- window manager
- session manager
- manipulation of shared resources
- device color characterisation.

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

2.2.1 Evolution of the Conventions

In the interests of timely acceptance, the **ICCCM** specification covers only a minimal set of required conventions. These conventions will be added to and updated as appropriate, based on the experiences of the X Consortium.

2.2.2 Structure of this Specification

The mapping of material contained in this specification to the MIT document is as follows:

MIT Chapter	Subject	X/Open Location
	New X/Open material	2.1
1	Overview	2.2
1.2	Atoms	2.3
2	Peer-to-peer Communication via Selections	3
3	Peer-to-peer Communication via Cut-buffers	4
4	Client-to-Window Manager Communication	5
5	Communicating with Session Managers	6
6	Manipulation of Shared Resources	7
7	Resource Manager Conventions	-
8	Conclusion	-

2.3 Atoms

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

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

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

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

2.3.1 Predefined Atoms

The protocol specifies a number of atoms as being predefined:

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

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

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

2.3.2 Naming Conventions

The built-in atoms are composed of upper-case ASCII characters with the logical words separated by an underscore character (_); 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 organisations, 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 upper case, special quoting conventions would have to be used.

---

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.

2.3.3 Semantics

The core protocol imposes no semantics on atoms except as they are used in FONTPROP structures. For further information on FONTPROP semantics, Chapter 11 on page 87.

2.3.4 Name Spaces

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

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

Chapter 3

Peer-to-Peer Communication by Means of Selections

Selections are the primary mechanism that X11 defines for the exchange of information between clients; for example, by cutting and pasting between windows. Note that there can be an arbitrary number of selections (each named by an atom) and that they are global to the server. The choice of an atom is discussed in Section 3.6 on page 21. Each selection is owned by a client and is attached to a window.

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

- the name of the selection
- the name of a property
- a window
- the atom representing the data type required.

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

- convert the contents of the selection to the requested data type
- place this data in the named property on the named window
- send the requestor an event to let it know the property is available.

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

Note that all data transferred between an owner and a requestor must usually go by means of the server in an X11 environment. A client cannot assume that another client can open the same files or even communicate directly. The other client may be talking to the server by means of a completely different networking mechanism (for example, one client might be DECnet and the other TCP/IP). Thus, passing indirect references to data (such as filenames, host names and port numbers, and so on) is permitted only if both clients specifically agree.

3.1 Acquiring Selection Ownership

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

SetSelectionOwner

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

The client should set the specified selection to the atom that represents the selection, set the specified owner to some window that the client created, and set the specified time to some time between the current last-change time of the selection concerned and the current server time. This time value usually will be obtained from the timestamp of the event that triggers the

acquisition of the selection. Clients should not set the time value to *CurrentTime*, because if they do so, they have no way of finding when they gained ownership of the selection. Clients must use a window they created so that requestors can route events to the owner of the selection.[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.*

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

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

GetSelectionOwner

selection: ATOM

=>

owner: WINDOW or *None*

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

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

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

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

3.2 Responsibilities of the Selection Owner

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

SelectionRequest

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

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

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

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

Otherwise, the owner should use the target to decide the form into which the selection should be converted. If the selection cannot be converted into that form, however, the owner should refuse the *SelectionRequest*, as previously described.

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

Convention: *All properties used to reply to SelectionRequest events must be placed on the requestor window.*

In either case, if the data comprising the selection cannot be stored on the requestor window (for example, because the server cannot provide sufficient memory), the owner must refuse the *SelectionRequest*, as previously described. See also Section 3.5 on page 20.

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

SelectionNotify

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

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

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

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 3.4 on page 18) or in properties with type MULTIPLE.*

A selection owner will often need confirmation that the data comprising the selection has actually been transferred. (For example, if the operation has side effects on the owner's internal data structures, these should not take place until the requestor has indicated that it has successfully received the data.) Owners should express interest in *PropertyNotify* events for the specified requestor window and wait until the property in the *SelectionNotify* event has been deleted before assuming that the selection data has been transferred.

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

SelectionClear

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

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

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

3.3 Giving Up Selection Ownership

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

3.3.1 Voluntarily Giving Up Selection Ownership

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

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

3.3.2 Forcibly Giving Up Selection Ownership

If a client gives up ownership of a selection or if some other client executes a *SetSelectionOwner* for it and thus reassigns it forcibly, the previous owner will receive a *SelectionClear* event. For the definition of a *SelectionClear* event, see Section 3.2 on page 15. The timestamp is the time the selection changed hands. The specified owner is the window that was specified by the current owner in its *SetSelectionOwner* request.

3.4 Requesting a Selection

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

ConvertSelection

selection, *target*: ATOM
property: ATOM or *None*
requestor: WINDOW
time: TIMESTAMP or *CurrentTime*

The selection argument specifies the particular selection involved, and the target argument specifies the required form of the information. For information about the choice of suitable atoms to use, see Section 3.6 on page 21. The requestor should set the requestor argument to a window that it created; the owner will place the reply property there. The requestor should set the time argument to the timestamp on the event that triggered the request for the selection value. Note that clients should not specify *CurrentTime*.

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

The requestor should set the property argument to the name of a property that the owner can use to report the value of the selection. Note that the requestor of a selection need not know the client that owns the selection or the window it is attached to.

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

Convention: *Requestors should not use None for the property argument of a ConvertSelectionrequest.*

Convention: *Owners receiving ConvertSelection requests with a property argument of Nonearetalkingtoanobsolete They should choose the target atom as the property name to be used for the reply.*

The result of the *ConvertSelection* request is that a *SelectionNotify* event will be received. For the definition of a *SelectionNotify* event, see Section 3.2 on page 15.

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

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

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

GetProperty

```
window: WINDOW
property: ATOM
type: ATOM or AnyPropertyType
long-offset, long-length: CARD32
delete: BOOL

=>

type: ATOM or None
format: {0, 8, 16, 32}
bytes-after: CARD32
value: LISTofINT8 or LISTofINT16 or LISTofINT32
```

When using *GetProperty* to retrieve the value of a selection, the property argument should be set to the corresponding value in the *SelectionNotify* event. Because the requestor has no way of knowing beforehand what type the selection owner will use, the type argument should be set to *AnyPropertyType*. Several *GetProperty* requests may be needed to retrieve all the data in the selection; each should set the long-offset argument to the amount of data received so far, and the size argument to some reasonable buffer size (see Section 3.5 on page 20). 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 Section 3.7 on page 26), the requestor should delete the property in the *SelectionNotify* request by using a *GetProperty* request with the delete argument set to *True*. As previously discussed, the owner has no way of knowing when the data has been transferred to the requestor unless the property is removed.

Convention: *The requestor must delete the property named in the SelectionNotify once all the data has been retrieved. The requestor should invoke either DeleteProperty or GetProperty (delete=True) after it has successfully retrieved all the data in the selection. For further information, see Section 3.5 on page 20.*

3.5 Large Data Transfers

Selections can get large, which poses two problems:

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

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

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

Convention: *Any client using SetSelectionOwnertoacquireselectionownershipshould to process Alloc errors in property change requests. For clients using Xlib, this involves using the XSetErrorHandler function to override the default handler.*

Convention: *A selection owner must confirm that no Allocerroroccurredwhilestoringthe before replying with a confirming SelectionNotify event.*

Convention: *When storing large amounts of data (relative to maximum-request-size), clients should use a sequence of ChangeProperty (mode=Append) requests for reasonable quantities of data. This avoids locking servers up and limits the waste of data an Alloc error would cause.*

Convention: *If an Alloc error occurs during the storing of the selection data, all properties stored for this selection should be deleted and the ConvertSelection request should be refused.*

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

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

3.6 Use of Selection Atoms

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

3.6.1 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.

3.6.1.1 The PRIMARY Selection

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

3.6.1.2 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 want to disturb it.

3.6.1.3 The CLIPBOARD Selection

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

- It should assert ownership of the CLIPBOARD.
- If it succeeds in acquiring ownership, it should be prepared to respond to a request for the contents of the CLIPBOARD in the usual way (retaining the data to be able to return it). The request may be generated by the clipboard client described below.
- If it fails to acquire ownership, a cutting client should not actually perform the cut or provide feedback that would suggest that it has actually transferred data to the clipboard.

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

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

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

- It should assert ownership of the CLIPBOARD selection and reassert it any time the clipboard data changes.
- If it loses the selection (because another client has some new data for the clipboard), it should:
  - — Obtain the contents of the selection from the new owner by using the timestamp in the *SelectionClear* event.
  - — Attempt to reassert ownership of the CLIPBOARD selection by using the same timestamp.
  - — Restart the process using a newly acquired timestamp if this attempt fails. This timestamp should be obtained by asking the current owner of the CLIPBOARD selection to convert it to a TIMESTAMP. If this conversion is refused or if the same timestamp is received twice, the clipboard client should acquire a fresh timestamp in the usual way (for example, by a zero-length append to a property).
- It should respond to requests for the CLIPBOARD contents in the usual way.

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

Stability	If the cutting client were to crash or terminate, the clipboard value would still be available.
Feedback	The clipboard client can display the contents of the clipboard.
Simplicity	A client deleting data does not have to retain it for so long, thus reducing the chance of race conditions causing problems.

The reasons not to run the clipboard client include:

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

3.6.2 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, the following table contains those that have been suggested so far.

Atom	Type	Data Received
TARGETS	ATOM	A list of valid target atoms.
MULTIPLE	ATOM_PAIR	(See the following discussion.)
TIMESTAMP	INTEGER	The timestamp used to acquire the selection.
STRING	STRING	ISO Latin-1 (+TAB+NEWLINE) text
COMPOUND_TEXT	COMPOUND_TEXT	Compound Text
TEXT	TEXT	The text in the owner's choice of encoding.
LIST_LENGTH	INTEGER	The number of disjoint parts of the selection.
PIXMAP	DRAWABLE	A list of pixmap IDs.
DRAWABLE	DRAWABLE	A list of drawable IDs.
BITMAP	BITMAP	A list of bitmap IDs.
FOREGROUND	PIXEL	A list of pixmap values.
BACKGROUND	PIXEL	A list of pixel values.
COLORMAP	COLORMAP	A list of colormap IDs.
ODIF	TEXT	ISO Office Document Interchange Format
OWNER_OS	TEXT	The operating system of the owner client.
FILE_NAME	TEXT	The full path name of a file.
HOST_NAME	TEXT	See WM_CLIENT_MACHINE.
CHARACTER_POSITION	SPAN	The start and end of the selection in bytes.
LINE_NUMBER	SPAN	The start and end line numbers.
COLUMN_NUMBER	SPAN	The start and end column numbers.
LENGTH	INTEGER	The number of bytes in the selection.
USER	TEXT	The name of the user running the owner.
PROCEDURE	TEXT	The name of the selected procedure.
MODULE	TEXT	The name of the selected procedure.
PROCESS	INTEGER, TEXT	The process ID of the owner.
TASK	INTEGER, TEXT	The task ID of the owner.
CLASS	TEXT	See WM_CLASS.
NAME	TEXT	See WM_NAME.
CLIENT_WINDOW	WINDOW	A top-level window of the owner.
DELETE	NULL	See DELETE.
INSERT_SELECTION	NULL	See INSERT_SELECTION.
INSERT_PROPERTY	NULL	See INSERT_PROPERTY.

It is expected that this table will grow over time.

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

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

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

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

— The owner should reply with a *SelectionNotify* only when all the requested conversions have been performed.

— If the owner fails to convert the target used by an atom in the MULTIPLE property, it should replace that atom in the property with *None*.

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

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

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

3.6.3 Selection Targets with Side Effects

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

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

Convention: *Targets with side effects should return no information (that is, they should have 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.*

Note: 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’’.

3.6.3.1 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.

3.6.3.2 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 should insert it at the location of the selection for which it got the INSERT_SELECTION request (whatever doing so means for its internal data structures).

3.6.3.3 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).

3.7 Use of Selection Properties

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

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

In all cases, a request for conversion to a target should return either a property of one of the types listed in the previous table for that 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 (for example, side-effect targets) will be of length zero irrespective of the number of disjoint selection parts. In the case of fixed-size items, the requestor may determine the number of items by the property size. For variable-length items such as text, the separators are listed in the following table:

Type Atom	Format	Separator
STRING	8	Null
COMPOUND_TEXT	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

It is expected that this table will grow over time.

3.7.1 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 the table in Section 3.6.2 on page 22, the word TEXT (in the Type column) is used to indicate one of the registered encoding names. The type would not actually be TEXT; it would be STRING or some other ATOM naming the encoding chosen by the owner.

STRING as a type or a target specifies the ISO Latin-1 character set plus the control characters TAB (octal 11) and NEWLINE (octal 12). The spacing interpretation of TAB is context-

dependent. Other ASCII control characters are explicitly not included in STRING at the present time.

COMPOUND_TEXT as a type or a target specifies the Compound Text interchange format; see the Chapter 13 on page 107.

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

3.7.2 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, which represents a lower bound on the number of bytes of data in the selection. The requestor and the selection owner transfer the data in the selection in the following manner.

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

The selection owner then:

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

The selection requestor:

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

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

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.

3.7.3 DRAWABLE Properties

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

- foreground pixel
- background pixel
- colormap ID.

In general, requestors converting into targets whose returned type in the table in Section 3.6.2 on page 22 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.

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

Chapter 4

Peer-to-Peer Communication by Means of Cut Buffers

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

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

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

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

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

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

Chapter 5

Client-to-Window Manager Communication

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

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

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

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

For example, the writer of a client application may assume the application is so important that its window should be the top window. If so, then perhaps it should be run under the control of a window manager that recognises ''important'' windows through some private protocol and ensures that they are on top. However, imagine, for example, that the ''important'' client is being debugged. Then, ensuring that it is always on top is no longer the appropriate window management policy, and it should be run under a window manager that allows other windows (for example, the debugger) to appear on top.

5.1 Actions of Client

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 the following:

- hinting to the window manager about the resources they would like to obtain
- cooperating with the window manager by accepting the resources they are allocated even if they are not those requested
- being prepared for resource allocations to change at any time.

5.1.1 Creating a Top-level Window

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

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

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

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

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

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

- Normal
- Iconic
- Withdrawn.

Newly created windows start in the Withdrawn state. Transitions between states happen when the top-level window is mapped and unmapped and when the window manager receives certain messages. For further details, see Section 5.1.2.4 on page 35 and Section 5.1.4 on page 39.

5.1.2 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 the behaviour that the client desires. Window managers will assume values they find convenient for any of these properties that are not supplied; clients that depend on particular values must explicitly supply them. The window manager will not change properties written by the client.

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

if it is shut down and restarted, or if the session needs to be shut down and restarted by the session manager.

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

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 next sections describe each of the properties the clients need to set, in turn. They are summarised in the table in Section 5.3 on page 54.

5.1.2.1 WM_NAME Property

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

The encoding used for this string (and all other uninterpreted string properties) is implied by the type of the property. The type atoms to be used for this purpose are described in Section 3.7.1 on page 26.

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

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

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

5.1.2.2 WM_ICON_NAME Property

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

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

5.1.2.3 WM_NORMAL_HINTS Property

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

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

The WM_SIZE_HINTS.flags bit definitions are as follows:

Name	Value	Field
USPosition	1	User-specified x, y
USSize	2	User-specified width, height
PPosition	4	Program-specified position
PSize	8	Program-specified size
PMinSize	16	Program-specified minimum size
PMaxSize	32	Program-specified maximum size
PResizeInc	64	Program-specified resize increments
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 the Withdrawn state) was specified by the user, the client should set the *USPosition* and *USSize* flags, which allow a window manager to know that the user specifically asked where the window should be placed or how the window should be sized and that further interaction is superfluous. To indicate that it was specified by the client without any user involvement, the client should set *PPosition* and *PSize*.

The size specifiers refer to the width and height of the client's window excluding borders. The 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 for non-negative integers *i* and *j*:

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

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

The min_aspect and max_aspect fields are fractions with the numerator first and the denominator second, and they allow a client to specify the range of aspect ratios it prefers.

5.1.2.4 WM_HINTS Property

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

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

The WM_HINTS.flags bit definitions are as follows:

Name	Value	Field
InputHint	1	input
StateHint	2	initial_state
IconPixmapHint	4	icon_pixmap
IconWindowHint	8	icon_window
IconPositionHint	16	icon_x & icon_y
IconMaskHint	32	icon_mask
WindowGroupHint	64	window_group
MessageHint	128	This bit is obsolete.

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 5.1.7 on page 43).

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 5.1.4 on page 39. 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 the Withdrawn state, as shown in the following table:

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

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

- One of the sizes specified in the WM_ICON_SIZE property on the root if it exists (see Section 5.1.3.2 on page 38).
- 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 non-rectangular.

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

Clients that need more capabilities from the icons than a simple two-color bitmap should use icon windows. Rules for clients that do are set out in Section 5.1.9 on page 46.

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 that exists only for that purpose; a placeholder group leader of this kind would never be mapped either by the client or by the window manager.*

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,[4] rather than the WM_PROTOCOLS mechanism of Section 5.1.2.7 on page 38.

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

5.1.2.5 WM_CLASS Property

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

The two strings, respectively, are:

- A string that names the particular instance of the application to which the client that owns this window belongs. Resources that are specified by instance name override any resources that are specified by class name. Instance names can be specified by the user in an operating system-specific manner. On POSIX-conformant systems, the following conventions are used:
  - If "–name NAME" is given on the command line, NAME is used as the instance name.
  - Otherwise, if the environment variable RESOURCE_NAME is set, its value will be used as the instance name.
  - Otherwise, the trailing part of the name used to invoke the program (argv[0] stripped of any directory names) is used as the instance name.
- A string that names the general class of applications to which the client that owns this window belongs. Resources that are specified by class apply to all applications that have the same class name. Class names are specified by the application writer. Examples of commonly used class names include: "Emacs", "XTerm", "XClock", "XLoad", and so on.

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

5.1.2.6 WM_TRANSIENT_FOR Property

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

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

5.1.2.7 WM_PROTOCOLS Property

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

All the protocols in which a client can volunteer to take part involve the window manager sending the client a *ClientMessage* event and the client taking appropriate action. For details of the contents of the event, see Section 5.2.8 on page 52. In each case, the protocol transactions are initiated by the window manager.

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

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

Protocol	Section	Purpose
WM_TAKE_FOCUS	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.

It is expected that this table will grow over time.

5.1.2.8 WM_COLORMAP_WINDOWS Property

The WM_COLORMAP_WINDOWS property (of type WINDOW) on a top-level window is a list of the IDs of windows that may need colormaps installed that differ from the colormap of the top-level window. The window manager will watch this list of windows for changes in their colormap attributes. The top-level window is always (implicitly or explicitly) on the watch list. For the details of this mechanism, see Section 5.1.8 on page 45.

5.1.3 Window Manager Properties

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

5.1.3.1 WM_STATE Property

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 6.1.1.3 on page 56 for more details.

5.1.3.2 WM_ICON_SIZE Property

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

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

For more details see the **Xlib - C Language Binding** specification.

5.1.4 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, whose semantics are as follows:

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 are 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 the Withdrawn state. Once the window has been provided with suitable properties, the client is free to change its state as follows:[5]

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 later in this section.

Normal → Withdrawn
The client should unmap the window and follow it with a synthetic *UnmapNotify* event as described later in this section.[6]

Iconic → Normal
The client should map the window. The contents of WM_HINTS.initial_state are irrelevant

5. The conventions described in earlier drafts of the ICCCM 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.

6. 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*.

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 the Withdrawn state, the client will know that the window is in the Normal state if it is mapped and that the window is in the Iconic state if it is not mapped. It may select for *StructureNotify* events on the top-level window, and it will receive an *UnmapNotify* event when it moves to the Iconic state and a *MapNotify* event when it moves to the Normal state. This implies that a reparenting window manager will unmap the top-level window as well as the parent window when changing to the Iconic state.

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 the Withdrawn state, a synthetic *UnmapNotify* event, in addition to unmapping the window itself, must be sent by using a *SendEvent* request with the following arguments:

Argument	Value
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 the Normal to the 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[7]
- Format = 32
- Data[0] = IconicState[8]

Other values of data[0] are reserved for future extensions to these conventions.[9] The parameters of the *SendEvent* event should be those described for the synthetic *UnmapNotify* event.

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

7. 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*).

8. We use the notation data[n] to indicate the nth 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.

9. The format of this *ClientMessage* event does not match the format of *ClientMessages* in Section 5.2.8 on page 52. This is because they are sent by the window manager to clients, and this is sent by clients to the window manager.

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

5.1.5 Configuring the Window

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

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

The coordinate system in which the location is expressed is that of the root (irrespective of any reparenting that may have occurred). The border width to be used and win_gravity position hint to be used are those most recently requested by the client. Client configure requests are interpreted by the window manager in the same manner as the initial window geometry mapped from the Withdrawn state, as described in Section 5.1.2.3 on page 33. Clients must be aware that there is no guarantee that the window manager will allocate them the requested size or location and must be prepared to deal with any size and location. If the window manager decides to respond to a *ConfigureRequest* request by:

- Not changing the size or location of the window at all.

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

- Moving the window without resizing it.

  A client will receive a synthetic *ConfigureNotify* event following the move that describes 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 that describes this change because the window manager may have reparented the top-level window. If the client does receive a real event, the synthetic event will follow the real one.

- Resizing the window (whether or not it is moved).

  A client that has selected for *StructureNotify* events 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. The coordinates will reflect the actual border width of the window (which the window manager may have changed). The *TranslateCoordinates* request can be used to convert the coordinates if required.

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

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

information. Other window managers will allow the top-level windows border to be visible.

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

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

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

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

Argument	Value
destination:	The root
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 field of both real and synthetic *ConfigureNotify* events that they receive on their non-override-redirect top-level windows because they cannot be guaranteed to contain useful information.

5.1.6 Changing Window Attributes

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

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

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

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

- Clients are free to set the save-under hint on their top-level windows, but they must be aware that the hint may be overridden by the window manager.
- Windows, in effect, have per-client event masks, and so, clients may select for whatever events are convenient irrespective of any events the window manager is selecting for. There are some events for which only one client at a time may select, but the window manager should not select for them on any of the client's windows.
- Clients can set override-redirect on top-level windows but are encouraged not to do so except as described in Section 5.1.10 on page 47 and Section 5.2.9 on page 53.

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

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

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

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

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

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

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

- *ButtonPress*
- *ButtonRelease*
- Passive-grabbed *KeyPress*
- Passive-grabbed *KeyRelease*.

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

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

Windows with the atom WM_TAKE_FOCUS in their WM_PROTOCOLS property may receive a *ClientMessage* event from the window manager (as described in Section 5.2.8 on page 52) 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. For further information, see Section 5.2.7 on page 52.

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

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

Note: 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 behaviour of the input focus if the window the focus has been set to becomes not viewable. The value can be any of the following:

Parent In general, clients should use this value when assigning focus to one of their subwindows. Unmapping the subwindow will cause focus to revert to the parent, which is probably what you want.

PointerRoot Using this value with a click-to-type focus management policy leads to race conditions because the window becoming unviewable may coincide with the window manager deciding to move the focus elsewhere.

None Using this value causes problems if the window manager reparents the window, as most window managers will, and then crashes. The input focus will be *None*, and there will probably be no way to change it.

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

Convention: *Clients that invoke a SetInputFocus request should set the revert-to argument to Parent.*

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

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

5.1.8 Colormaps

The window manager is responsible for installing and uninstalling colormaps.[10] Clients provide the window manager with hints as to which colormaps to install and uninstall, but clients 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 requests.*

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 one of the following:

- a priority ordered list of the colormaps they want installed
- 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, thus, permitting visual-dependent colormap installation policies.
- It allows window managers to select for *VisibilityChange* events 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 its ID 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. The window manager will install the colormap for them.

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

---

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

Top-level windows that have subwindows or override-redirect pop-up windows whose colormap requirements differ from the top-level window should have a WM_COLORMAP_WINDOWS property. This property contains a list of IDs for windows whose colormaps the window manager should attempt to have installed when, in the course of its individual colormap focus policy, it assigns the colormap focus to the top-level window (see Section 5.1.2.8 on page 38). 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 either can have their own WM_COLORMAP_WINDOWS property or can 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 so-called ‘‘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 less precise words, the min-installed-maps most recently installed maps are guaranteed to be installed. Min-installed-maps 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 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, will attempt to maximise the number of colormaps from the head of the WM_COLORMAP_WINDOWS list that is installed.

5.1.9 Icons

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

- A string in WM_ICON_NAME.

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

- A *Pixmap* into the icon_pixmap field of the WM_HINTS property and possibly another into the icon_mask field .

  The window manager is expected to display the pixmap masked by the mask. The pixmap should be one of the sizes found in the WM_ICON_SIZE property on the root. If this property is not found, the window manager is unlikely to display icon pixmaps. Window managers usually will clip or tile pixmaps that do not match WM_ICON_SIZE.

- A window into the icon_window field of the WM_HINTS property.

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

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

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

Clients should observe the following conventions about their icon windows:

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 or select for ResizeRedirect events on them.*

Convention: *Clients must not depend on being able to receive input events by means of their icon windows.*

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

Convention: *Clients must select for Exposure events on their icon window and repaint it when requested.*

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

Window managers will ignore any WM_NAME, WM_ICON_NAME, WM_NORMAL_HINTS, WM_HINTS, WM_CLASS, WM_TRANSIENT_FOR, WM_PROTOCOLS 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.

5.1.10 Pop-up Windows

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

1. They can create and map another normal top-level window, which will get decorated and managed as normal by the window manager. See the discussion of window groups that follows.
2. 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.

3. 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.

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

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

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

5.2 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.

5.2.1 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 as follows:

- The parent value returned by a *QueryTree* request will no longer be the value supplied to the *CreateWindow* request that created the reparented window. There should be no need for the client to be aware of the identity of the window to which the top-level window has been reparented. In particular, a client that wishes to create further top-level windows should continue to use the root as the parent for these new windows.
- The server will interpret the (x,y) coordinates in a *ConfigureWindow* request in the new parent's coordinate space. In fact, they usually will not be interpreted by the server because a reparenting window manager usually will have intercepted these operations (see Section 5.2.2 on page 50). Clients should use the root coordinate space for these requests (see Section 5.1.5 on page 41).
- *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 5.1.5 on page 41).
- The (x,y) coordinates returned by a *GetGeometry* request are in the parent's coordinate space and are thus not directly useful after a reparent operation.
- A background of *ParentRelative* will have unpredictable results.
- A cursor of *None* will have unpredictable results.

Clients that want to be notified when they are reparented can select for *StructureNotify* events on their top-level window. They will receive a *ReparentNotify* event if and when reparenting takes place.

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

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

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

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

5.2.2 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 cannot assume that any redirectable request is actually performed when the request is issued or is actually performed at all. For example, the following is incorrect because the *MapWindow* request may be intercepted and the *PolyLine* output made to an unmapped window:

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

The client must wait for an *Expose* event before drawing in the window.[11]

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

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

The requests that 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 5.1.10 on page 47) and while responding to *ResizeRequest* events (see Section 5.2.9 on page 53).

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

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

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

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

5.2.3 Window Move

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

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

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

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

5.2.4 Window Resize

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

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

5.2.5 Iconify and De-iconify

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

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

5.2.6 Colormap Change

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

Note: 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.

5.2.7 Input Focus

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

- Set the input field of WM_HINTS to *True* and actually have the input focus in one of their top-level windows.
- Set the input field of WM_HINTS to *False* and have received a suitable event as described in Section 5.1.7 on page 43.
- Have received a WM_TAKE_FOCUS message as described in Section 5.1.7 on page 43.

Clients should not warp the pointer in an attempt to transfer the focus; they should set the focus and leave the pointer alone. For further information, see Section 7.2 on page 62.

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

SetInputFocus

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

Convention: *Clients that use a SetInputFocus request must set the time argument to the timestamp of the event that caused them to make the attempt. This cannot be a FocusIn event because they do not have timestamps. Clients may also acquire the focus without a corresponding EnterNotify event. Clients must not use CurrentTime for the time argument.*

Convention: *Clients that use a SetInputFocus request to set the focus to one of their windows must set the revert-to field to Parent.*

5.2.8 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 5.1.2.7 on page 38). For all protocols, the *ClientMessage* events have the following:

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

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

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

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

5.2.9 Redirecting Requests

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

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

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

Convention: *Clients receiving ResizeRequest events must respond by doing the following:*

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

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

5.3 Summary of Window Manager Property Types

The window manager properties are summarised in the following table (see also the **Xlib - C Language Binding** specification).

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

Chapter 6

Client-to-Session Manager Communication

The session manager's role 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.

6.1 Client Actions

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

1. Stateful clients should cooperate with the session manager by providing it with information it can use to restart them if that should become necessary.
2. Clients, typically those with more than one top-level window, whose server connection needs to survive the deletion of their top-level window should take part in the WM_DELETE_WINDOW protocol (see Section 6.2.2 on page 59).

6.1.1 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.

6.1.1.1 WM_COMMAND Property

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

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

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

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

compatibility.

A client with multiple top-level windows should ensure that exactly one of them has a WM_COMMAND with nonzero length. Zero-length WM_COMMAND properties can be used to reply to WM_SAVE_YOURSELF messages on other top-level windows but will otherwise be ignored (see Section 6.2.1 on page 58).

6.1.1.2 WM_CLIENT_MACHINE Property

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

6.1.1.3 WM_STATE Property

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 client's top-level windows can use this property to identify them. A client's top-level window is one that has override-redirect set to *False* and a WM_STATE property or that is 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 the ICCCM. The presence or absence of the property is all they need to know.

Suggested contents of the WM_STATE property are listed in the following table:

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

The following table lists the WM_STATE.state values:

State	Value
WithdrawnState	0
NormalState	1
IconicState	3

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

The icon field should either 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 one exists or *None* if one does not. Note that this window is not necessarily the same as the icon window that the client may have specified. It can be one of the following:

- the client's icon window
- a window that the window manager supplied and that contains the client's icon pixmap
- the least ancestor of the client's icon window (or of the window that contains the client's icon pixmap), which 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*, the

client should animate its icon window. 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 forthcoming *Window and Session Manager Conventions Manual*.

6.1.2 Termination

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

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

6.2 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.

6.2.1 Saving Client State

Clients that want to be warned when the session manager feels that they should save their internal state (for example, when termination impends) should include the atom WM_SAVE_YOURSELF in the WM_PROTOCOLS property on their top-level windows to participate in the WM_SAVE_YOURSELF protocol. They will receive a *ClientMessage* event as described in Section 5.2.8 on page 52 with the atom WM_SAVE_YOURSELF in its data[0] field.

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

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

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

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

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

Receiving WM_SAVE_YOURSELF on a window is, conceptually, a command to save the entire client state.[12]

12. 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.

6.2.2 Window Deletion

Clients, usually those with multiple top-level windows, whose server connection must survive the deletion of some of their top-level windows should include the atom WM_DELETE_WINDOW in the WM_PROTOCOLS property on each such window. They will receive a *ClientMessage* event as described in Section 5.2.8 on page 52 whose data[0] field is WM_DELETE_WINDOW.

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

- Either change the window's state to Withdrawn (as described in Section 5.1.4 on page 39) or destroy the window.
- Destroy any internal state associated with the window.

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

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

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

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

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

6.3 Summary of Session Manager Property Types

The session manager properties are listed in the following table:

Name	Type	Format
WM_CLIENT_MACHINE	TEXT	
WM_COMMAND	TEXT	
WM_STATE	WM_STATE	32

Chapter 7

Manipulation of Shared Resources

X11 permits clients to manipulate a number of shared resources; for example, the input focus, the pointer and colormaps. Conventions are required so that clients share resources in an orderly fashion.

7.1 The Input Focus

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

- locally active mode
- globally active mode.

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 to True.*

Convention: *Globally active clients should set the input focus to one of their windows only when they receive a button event and a passive-grabbed key event, or when they receive a WM_TAKE_FOCUS message. They should set the input field of the WM_HINTS structure to False.*

Convention: *In addition, clients should use the timestamp of the event that caused them to attempt to set the input focus as the time field on the SetInputFocus request, not CurrentTime.*

7.2 The Pointer

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

Convention: *Clients should not warp the pointer.*

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

7.3 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, therefore, are shared resources, and their use requires conventions.

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

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

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

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

Convention: *Clients should establish button and key grabs only on windows that they own.*

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

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

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

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

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

```
mode = ReplayPointer/Keyboard
```

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

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

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

7.4 Colormaps

This section prescribes the following:

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, it 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 requests.*

If your clients are *DirectColor* type applications, you should consult the **Xlib - C Language Binding** specification for conventions connected with sharing standard colormaps. They should look for and create the properties described there on the root window of the appropriate screen.

The contents of the RGB_COLOR_MAP type property are as follows:

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

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

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

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

7.5 The Keyboard Mapping

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

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

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

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

Client: ''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

Client: ''Adding Pause to the symbols on the Scroll Lock key: Confirm or Abort.''

User: Confirms

Client: Uses a *ChangeKeyboardMapping* request to add Pause to the keycode that already contains Scroll Lock and issues this request, ''Please paint Pause on the Scroll Lock key''.

Convention: *Clients should not use ChangeKeyboardMapping requests.*

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

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

7.6 The Modifier Mapping

X11 supports eight modifier bits of which three are preassigned to Shift, Lock and Control. Each modifier bit is controlled by the state of a set of keys, and these sets are specified in a table accessed by *GetModifierMapping* and *SetModifierMapping* requests. This table is a shared resource and requires conventions.

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

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

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

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

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 de-assign obsolete modifier mappings by hand.*

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

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

Chapter 8

Device Color Characterisation

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

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

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

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

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

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

A 3x3 matrix M (and it's inverse, M^{-1}) which convert between XYZ and RGB intensity ($RGB_{intensity}$):

$$RGB_{intensity} = M \times XYZ$$

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

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

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

8.1 XYZ RGB Conversion Matrices

Because of the limited dynamic range of both XYZ and RGB intensity, these matrices will be encoded using a fixed point representation of a 32-bit 2s complement number scaled by 2^{27}, giving a range of –16 to 16–ε, where $\varepsilon = 2^{-37}$.

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

This will be encoded as:

XDCCC_LINEAR_RGB_MATRICES property contents

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

8.2 Intensity RGB value Conversion

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

0 RGB value / RGB intensity level pairs

1 RGB intensity ramp.

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

The first representation is an array of RGB value / intensity level pairs, with the RGB values in strictly increasing order. When converting, the client must linearly interpolate between adjacent entries in the table to compute the desired value. This is to allow the server to perform gamma correction itself and encode that fact in a short 2 element correction table. The intensity will be encoded as an unsigned number to be interpreted as a value between 0 and 1 (inclusive). The precision of this value will depend on the format of the property in which it is stored (8, 16 or 32 bits). For 16 and 32 bit formats, the RGB value will simply be the value stored in the property. When stored in 8-bit format, the RGB value can be computed from the value in the property by:

$$RGB_{value} = \frac{Property\ Value \times 65535}{255}$$

Because the three electron guns in the device may not be exactly alike in response characteristics, it is necessary to allow for three separate tables, one each for red, green and blue. So, each table will be preceded by the number of entries in that table, and the set of tables will be preceded by the number of tables. When 3 tables are provided, they will be in red, green, blue order.

This will be encoded as:

XDCCC_LINEAR_RGB_CORRECTION property contents for type 0 correction

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

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

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

$$RGB_{value} = \frac{Array\ Index \times 65535}{Array\ Size - 1}$$

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

This will be encoded as:

XDCCC_LINEAR_RGB_CORRECTION property contents for type 1 correction

Field	*Type*	*Comments*
VisualID0	CARD	Most significant portion of VisualID
VisualID1	CARD	(exists iff property format is 8)
VisualID2	CARD	(exists iff property format is 8)
VisualID3	CARD	Least significant (exists iff property format is 8 or 16)
type	CARD	1 for this type of correction
count	CARD	number of tables following (either 1 or 3)
length	CARD	number of elements - 1 following in this table
intensity	CARD	Interpret as a number 0 <= intensity <= 1
...	...	Total of *length+1* intensity elements
lengthg	CARD	number of elements - 1 following in this table (iff *count* is 3)
intensity	CARD	Interpret as a number 0 <= intensity <= 1
...	...	Total of *lengthg+1* intensity elements
lengthb	CARD	number of elements - 1 following in this table (iff *count* is 3)
intensity	CARD	Interpret as a number 0 <= intensity <= 1
...	...	Total of *lengthb+1* intensity elements

Part 2:

X Logical Font Description (XLFD)

X/Open Company Ltd.

Chapter 9

Introduction to XLFD

It is a requirement that X client applications must be portable across server implementations, with very different file systems, naming conventions and font libraries. However, font access requests, as defined by the **X Window System Protocol** specification, neither specify server-independent conventions for font names nor provide adequate font properties for logically describing typographic fonts.

X clients must be able to dynamically determine the fonts available on any given server so that understandable information can be presented to the user or that intelligent font fallbacks can be chosen. It is desirable for the most common queries to be accomplished without the overhead of opening each font and inspecting font properties, by means of simple *ListFonts* requests. For example, if a user selected a Helvetica typeface family, a client application should be able to query the server for all Helvetica fonts and present only those setwidths, weights, slants, point sizes and character sets available for that family.

This document gives a standard logical font description (hereafter referred to as XLFD) and the conventions to be used in the core protocol so that clients can query and access screen type libraries in a consistent manner across all X servers. In addition to completely specifying a given font by means of its *FontName*, the XLFD also provides for a standard set of key *FontProperties* that describe the font in more detail.

The XLFD provides an adequate set of typographic font properties, such as CAP_HEIGHT, X_HEIGHT, RELATIVE_SETWIDTH, for publishing and other applications to do intelligent font matching or substitution when handling documents created on some foreign server that use potentially unknown fonts. In addition, this information is required by certain clients to position subscripts automatically and determine small capital heights, recommended leading, word-space values, and so on.

9.1 Status

This specification is syntactically and semantically complete.

All the facilities specified in this specification are mandatory.

There are no parts of this specification which are optional.

Internationalisation

The X Window System is 8-bit transparent. Any 8-bit or 16-bit codeset may be used in the font and text calls. In addition, 8-bit codesets may be used in all strings including filenames, atom names and color names.

9.1.1 Structure of this Specification

XLFD is divided into two basic components; the *FontName*, which gives all font information needed to uniquely identify a font in X Protocol requests (such as *OpenFont* and *ListFonts*) and a variable list of optional *FontProperties* that describe a font in more detail.

Chapter 10 on page 77 describes the *FontName*, which is used in font queries and returned as data in certain X Protocol requests. The *FontName* is also specified as the data value for the *FONT* item in the **BDF** specification.

Chapter 11 on page 87 describes the *FontProperties*, which are supplied on a font-by-font basis, and are returned as data in certain X Protocol requests as part of the *XFontStruct* data structure. The *FontProperties* names and associated data values may also appear as items of the STARPROPERTIES...ENDPROPERTIES list in the BDF V2.1 specification (see the **BDF** specification).

The XLFD provides an adequate set of typographical font properties, such as CAP_HEIGHT, X_HEIGHT, RELATIVE_SETWIDTH, for publishing and other applications to do intelligent font matching or substitution when handling documents created on some foreign server using potentially unknown fonts. Some clients also need this information automatically to place subscripts, and to determine small capital heights, recommended leading, wordspace values, and so on.

Chapter 12 on page 103 indicates the elements of Xlib and the X Protocol that these conventions affect, and specify how to test conformance to the BDF.

The mapping of material contained in this specification to the MIT document is as follows:

MIT Chapter	Subject	X/Open Location
1	Introduction	-
-	New X/Open material	9.1
3	X Logical Font Description	9.2
-	New X/Open material	9.2
2	Requirements and Goals	9.3
3.1	Font Name	10
3.2	Font Properties	11
4	Affected Elements of Xlib and the X Protocol	12.1
5	BDF Conformance	12.2

9.2 Requirements and Goals

The XLFD meets the short and long-term goals to have a standard logical font description that:

- provides unique, descriptive font names that support simple pattern matching
- supports multiple font vendors, arbitrary character sets and encodings
- supports naming and instancing of scalable fonts
- is independent of X server and operating or file system implementations
- supports arbitrarily complex font matching or substitution
- is extensible.

9.2.1 Provide Unique and Descriptive Font Names

It should be possible to have font names that are long enough and descriptive enough to have a reasonable probability of being unique without inventing a new registration organisation. Resolution and size-dependent font masters, multi-vendor font libraries, and so on must be anticipated and handled by the font name alone.

The name itself should be structured to be amenable to simple pattern matching and parsing, thus, allowing X clients to restrict font queries to some subset of all possible fonts in the server.

9.2.2 Support Multiple Font Vendors and Character Sets

The font name and properties should distinguish between fonts that were supplied by different font vendors but that possibly share the same name. We anticipate a highly competitive font market where users will be able to buy fonts from many sources according to their particular requirements.

A number of font vendors deliver each font with all glyphs designed for that font, where charset mappings are defined by encoding vectors. Some server implementations may force these mappings to proprietary or standard charsets statically in the font data. Others may desire to perform the mapping dynamically in the server. Provisions must be made in the font name that allows a font request to specify or identify specific charset mappings in server environments where multiple charsets are supported.

9.2.3 Support Scalable Fonts

If a font source can be scaled to arbitrary size, it should be possible for an application to determine that fact from the font name, and the application should be able to construct a font name for any specific size.

9.2.4 Be Independent of X Server and Operating or File System Implementations

X client applications that require a particular font should be able to use the descriptive name without knowledge of the file system or other repository in use by the server. However, it should be possible for servers to translate a given font name into a file name syntax that it knows how to deal with, without compromising the uniqueness of the font name. This algorithm should be reversible (exactly how this translation is done is implementation-dependent).

9.2.5 Support Arbitrarily Complex Font Matching and Substitution

In addition to the font name, the XLFD should define a standard list of descriptive font properties, with agreed upon fallbacks for all fonts. This allows client applications to derive font-specific formatting or display data and to perform font matching or substitution when asked to handle potentially unknown fonts, as required.

9.2.6 Extensible

The XLFD must be extensible so that new and/or private descriptive font properties can be added to conforming fonts without making existing X client or server implementations obsolete.

Chapter 10

FontName

Each *FontName* is logically composed of two strings: a *FontNameRegistry* prefix that is followed by a *FontNameSuffix*. The *FontNameRegistry* is an x-registered-name (a name that has been registered with the X Consortium) that identifies the registration authority that owns the specified *FontNameSuffix* syntax and semantics.

All font names that conform to this specification are to use a *FontNameRegistry* prefix, which is defined to be the string ''–'' (that is, ISO 8859-1 HYPHEN -- Column/Row 02/13). All *FontNameRegistry* prefixes of the form: *+version–*, where the specified version indicates some future XLFD specification, are reserved by the X Consortium for future extensions to XLFD font names. If required, extensions to the current XLFD font name shall be constructed by appending new fields to the current structure, each delimited by the existing field delimiter. The availability of other *FontNameRegistry* prefixes or fonts that support other registries is server implementation-dependent.

In the **X Window System Protocol** specification, the *FontName* is required to be a string; hence, numeric field values are represented in the name as string equivalents. All *FontNameSuffix* fields are also defined as *FontProperties*; numeric property values are represented as signed or unsigned integers, as appropriate.

10.1 FontName Syntax

The *FontName* is a structured, parsable string (of type STRING8) whose Backus-Naur Form syntax description is as follows:

FontName ::=	XFontNameRegistry XFontNameSuffix | PrivFontNameRegistry PrivFontNameSuffix
XFontNameRegistry ::=	XFNDelim | XFNExtPrefix Version XFNDelim
XFontNameSuffix ::=	FOUNDRY XFNDelim FAMILY_NAME XFNDelim WEIGHT_NAME XFNDelim SLANT XFNDelim SETWIDTH_NAME XFNDelim ADD_ STYLE_NAME XFNDelim PIXEL_SIZE XFNDelim POINT_SIZE XFNDelim RESOLUTION_X XFNDelim RESOLUTION_Y XFNDelim SPACING XFNDelim AVERAGE_WIDTH XFNDelim CHARSET_REGISTRY XFNDelim CHARSET_ENCODING
Version ::=	STRING8 – the XLFD version that defines an extension to the font name syntax (for example, ''1.4'')
XFNExtPrefix ::=	OCTET – the value of ISO8859-1 PLUS (Column/Row 02/11)
XFNDelim ::=	OCTET – the value of ISO8859-1 HYPHEN (Column/Row 02/13)
PrivFontNameRegistry ::=	STRING8 – other than those strings reserved by XLFD
PrivFontNameSuffix ::=	STRING8

Field values are constructed as strings of ISO 8859-1 graphic characters, excluding the following:

- HYPHEN (02/13), the XLFD font name delimiter character
- QUESTION MARK (03/15) and ASTERISK (02/10), the X Protocol fontname wildcard characters.

Alphabetic case distinctions are allowed but are for human readability concerns only. Conforming X servers will perform matching on font name query or open requests independent of case. The entire font name string must have no more than 255 characters. It is recommended that clients construct font name query patterns by explicitly including all field delimiters to avoid unexpected results. Note that SPACE is a valid character of a *FontName* field; for example, the string "ITC Avant Garde Gothic" might be a FAMILY_NAME.

10.2 FontName Field Definitions

This section discusses the *FontName*:

- FOUNDRY field
- FAMILY_NAME field
- WEIGHT_NAME field
- SLANT field
- SETWIDTH_NAME field
- ADD_STYLE_NAME field
- PIXEL_SIZE field
- POINT_SIZE field
- RESOLUTION_X and RESOLUTION_Y fields
- SPACING field
- AVERAGE_WIDTH field
- CHARSET_REGISTRY and CHARSET_ENCODING fields.

10.2.1 FOUNDRY Field

FOUNDRY is an x-registered-name, the name or identifier of the digital type foundry that digitised and supplied the font data, or if different, the identifier of the organisation that last modified the font shape or metric information.

The reason this distinction is necessary is that a given font design may be licensed from one source (for example, ITC) but digitised and sold by any number of different type suppliers. Each digital version of the original design, in general, will be somewhat different in metrics and shape from the idealised original font data, because each font foundry, for better or for worse, has its own standards and practices for tweaking a typeface for a particular generation of output technologies or has its own perception of market needs.

It is up to the type supplier to register with the X Consortium a suitable name for this *FontName* field according to the registration procedures defined by the Consortium.

The X Consortium shall define procedures for registering foundry and other names and shall maintain and publish, as part of its public distribution, a registry of such registered names for use in XLFD font names and properties.

10.2.2 FAMILY_NAME Field

FAMILY_NAME is a string that identifies the range or ‘‘family’’ of typeface designs that are all variations of one basic typographic style. This must be spelled out in full, with words separated by spaces, as required. This name must be human-understandable and suitable for presentation to a font user to identify the typeface family.

It is up to the type supplier to supply and maintain a suitable string for this field and font property, to secure the proper legal title to a given name, and to guard against the infringement of other’s copyrights or trade marks. By convention, FAMILY_NAME is not translated. FAMILY_NAME may include an indication of design ownership if considered a valid part of the typeface family name.

The following are examples of FAMILY_NAME:

- Helvetica
- ITC Avant Garde Gothic
- Times
- Times Roman
- Bitstream Amerigo
- Stone.

10.2.3 WEIGHT_NAME Field

WEIGHT_NAME is a string that identifies the font's typographic weight, that is, the nominal blackness of the font, according to the FOUNDRY's judgement. This name must be human-understandable and suitable for presentation to a font user.

The interpretation of this field is somewhat problematic because the typographic judgement of weight has traditionally depended on the overall design of the typeface family in question; that is, it is possible that the DemiBold weight of one font could be almost equivalent in typographic feel to a Bold font from another family.

WEIGHT_NAME is captured as an arbitrary string because it is an important part of a font's complete human-understandable name. However, it should not be used for font matching or substitution. For this purpose, X client applications should use the weight-related font properties (RELATIVE_WEIGHT and WEIGHT) that give the coded relative weight and the calculated weight, respectively.

10.2.4 SLANT Field

SLANT is a code-string that indicates the overall posture of the typeface design used in the font. The encoding is as follows:

Code	English Translation	Description
''R''	Roman	Upright design.
''I''	Italic	Italic design, slanted clockwise from the vertical.
''O''	Oblique	Obliqued upright design, slanted clockwise from the vertical.
''RI''	Reverse Italic	Italic design, slanted counterclockwise from the vertical.
''RO''	Reverse Oblique	Obliqued upright design, slanted counterclockwise from the vertical.
''OT''	Other	Other.

The SLANT codes are for programming convenience only and usually are converted into their equivalent human-understandable form before being presented to a user.

10.2.5 SETWIDTH_NAME Field

SETWIDTH_NAME is a string that gives the font's typographic proportionate width; that is, the nominal width per horizontal unit of the font, according to the FOUNDRY's judgement.

As with WEIGHT_NAME, the interpretation of this field or font property is somewhat problematic, because the designer's judgement of setwidth has traditionally depended on the overall design of the typeface family in question. For purposes of font matching or substitution, X client applications should either use the RELATIVE_SETWIDTH font property that gives the relative coded proportionate width or calculate the proportionate width.

The following are examples of SETWIDTH_NAME:

- Normal
- Condensed
- Narrow
- Double Wide.

10.2.6 ADD_STYLE_NAME Field

ADD_STYLE_NAME is a string that identifies additional typographic style information that is not captured by other fields but is needed to identify the particular font.

ADD_STYLE_NAME is not a typeface classification field and is only used for uniqueness. Its use, as such, is not limited to typographic style distinctions.

The following are examples of ADD_STYLE_NAME:

- Serif
- Sans Serif
- Informal
- Decorated.

10.2.7 PIXEL_SIZE Field

PIXEL_SIZE is an unsigned integer-string typographic metric in device pixels that gives the body size of the font at a particular POINT_SIZE and RESOLUTION_Y. PIXEL_SIZE usually incorporates additional vertical spacing that is considered part of the font design. (Note, however, that this value is not necessarily equivalent to the height of the font bounding box.) PIXEL_SIZE is in the range zero to a very large number. Zero is used to indicate a scalable font.

PIXEL_SIZE usually is used by X client applications that need to query fonts according to device-dependent size, regardless of the point size or vertical resolution for which the font was designed.

10.2.8 POINT_SIZE Field

POINT_SIZE is an unsigned integer-string typographic metric in device-independent units that gives the body size for which the font was designed. This field usually incorporates additional vertical spacing that is considered part of the font design. (Note, however, that POINT_SIZE is not necessarily equivalent to the height of the font bounding box.) POINT_SIZE is expressed in decipoints (where points are as defined in the X Protocol or 72.27 points equal 1 inch) in the range zero to a very large number. Zero is used to indicate a scalable font.

POINT_SIZE and RESOLUTION_Y are used by X clients to query fonts according to device-independent size to maintain constant text size on the display regardless of the PIXEL_SIZE used for the font.

10.2.9 RESOLUTION_X and RESOLUTION_Y Fields

RESOLUTION_X and RESOLUTION_Y are unsigned integer-strings that give the horizontal and vertical resolution, measured in pixels or dots per inch (dpi), for which the font was designed. Horizontal and vertical values are required because a separate bitmap font must be designed for displays with very different aspect ratios (for example, 1:1, 4:3, 2:1, and so on).

The separation of pixel or point size and resolution is necessary because X allows for servers with very different video characteristics (for example, horizontal and vertical resolution, screen and pixel size, pixel shape, and so on) to potentially access the same font library. The font name, for example, must differentiate between a 14 point font designed for 75 dpi (body size of about 14 pixels) or a 14 point font designed for 150 dpi (body size of about 28 pixels). Further, in servers that implement some or all fonts as continuously scaled and scan-converted outlines, POINT_SIZE and RESOLUTION_Y will help the server to differentiate between potentially separate font masters for text, title and display sizes or for other typographic considerations.

10.2.9.1 SPACING Field

SPACING is a code-string that indicates the escapement class of the font; that is, monospace (fixed pitch), proportional (variable pitch) or charcell (a special monospaced font that conforms to the traditional data processing character cell font model). The encoding is as follows:

Code	English Translation	Description
‘‘P’’	Proportional	A font whose logical character widths vary for each glyph. Note that no other restrictions are placed on the metrics of a proportional font.
‘‘M’’	Monospaced	A font whose logical character widths are constant (that is, every glyph in the font has the same logical width). No other restrictions are placed on the metrics of a monospaced font.
‘‘C’’	CharCell	A monospaced font that follows the standard typewriter character cell model (that is, the glyphs of the font can be modeled by X clients as ‘‘boxes’’ of the same width and height that are imaged side-by-side to form text strings or top-to-bottom to form text lines. By definition, all glyphs have the same logical character width, and no glyphs have ‘‘ink’’ outside of the character cell. There is no kerning (that is, on a per character basis with positive metrics: 0 <= left-bearing <= right-bearing <= width; with negative metrics: width <= left-bearing <= right-bearing <= zero). Also, the vertical extents of the font do not exceed the vertical spacing (that is, on a per character basis: ascent <= font-ascent and descent <= font-descent). The cell height = font-descent + font-ascent, and the width = AVERAGE_WIDTH.

10.2.10 AVERAGE_WIDTH Field

AVERAGE_WIDTH is an unsigned integer-string typographic metric value that gives the unweighted arithmetic mean width of all glyphs in the font (measured in tenths of pixels). For monospaced and character cell fonts, this is the width of all glyphs in the font. AVERAGE_WIDTH is in the range zero to a very large number. Zero is used to indicate a scalable font.

10.2.11 CHARSET_REGISTRY and CHARSET_ENCODING Fields

The character set used to encode the glyphs of the font (and implicitly the font’s glyph repertoire), as maintained by the X Consortium character set registry. CHARSET_REGISTRY is an x-registered-name that identifies the registration authority that owns the specified encoding. CHARSET_ENCODING is a registered-name that identifies the coded character set as defined by that registration authority.

Although the X Protocol does not explicitly have any knowledge about character set encodings, it is expected that server implementors will prefer to embed knowledge of certain proprietary or standard charsets into their font library for reasons of performance and convenience. The CHARSET_REGISTRY and CHARSET_ENCODING fields or properties allow an X client font request to specify a specific charset mapping in server environments where multiple charsets are supported. The availability of any particular character set is font and server implementation-dependent.

To prevent collisions when defining character set names, it is recommended that CHARSET_REGISTRY and CHARSET_ENCODING name pairs be constructed according to the following conventions:

CharsetRegistry ::=	StdCharsetRegistryName | PrivCharsetRegistryName
CharsetEncoding ::=	StdCharsetEncodingName | PrivCharsetEncodingName
StdCharsetRegistryName ::=	StdOrganizationId StdNumber | StdOrganizationId StdNumber Dot Year
PrivCharsetRegistryName ::=	OrganizationId STRING8
StdCharsetEncodingName ::=	STRING8–numeric part number of referenced standard
PrivCharsetEncodingName ::=	STRING8
StdOrganizationId ::=	STRING8–the registered name or acronym of the referenced standard organisation
StdNumber ::=	STRING8–referenced standard number
OrganizationId ::=	STRING8–the registered name or acronym of the organisation
Dot ::=	‘‘.’’–ISO 8859-1 FULL STOP (Column / Row 2 / 14)
Year ::=	STRING8–numeric year (for example, 1989)

The X Consortium shall maintain and publish a registry of such character set names for use in X Protocol font names and properties as specified in XLFD.

The ISO Latin-1 character set shall be registered by the X Consortium as the CHARSET_REGISTRY-CHARSET_ENCODING value pair: ‘‘ISO8859-1’’.

10.3 Examples

The following examples of font names are derived from the screen fonts shipped with the MIT X distribution.

Font	X FontName
75 dpi Fonts	
Charter 12 pt	-Bitstream-Charter-Medium-R-Normal--12-120-75-75-P-68-ISO8859-1
Charter Bold 12 pt	-Bitstream-Charter-Bold-R-Normal--12-120-75-75-P-76-ISO8859-1
Charter Bold Italic 12 pt	-Bitstream-Charter-Bold-I-Normal--12-120-75-75-P-75-ISO8859-1
Charter Italic 12 pt	-Bitstream-Charter-Medium-I-Normal--12-120-75-75-P-66-ISO8859-1
Courier 8 pt	-Adobe-Courier-Medium-R-Normal--8-80-75-75-M-50-ISO8859-1
Courier 10 pt	-Adobe-Courier-Medium-R-Normal--10-100-75-75-M-60-ISO8859-1
Courier 12 pt	-Adobe-Courier-Medium-R-Normal--12-120-75-75-M-70-ISO8859-1
Courier 14 pt	-Adobe-Courier-Medium-R-Normal--14-140-75-75-M-90-ISO8859-1
Courier 18 pt	-Adobe-Courier-Medium-R-Normal--18-180-75-75-M-110-ISO8859-1
Courier 24 pt	-Adobe-Courier-Medium-R-Normal--24-240-75-75-M-150-ISO8859-1
Courier Bold 10 pt	-Adobe-Courier-Bold-R-Normal--10-100-75-75-M-60-ISO8859-1
Courier Bold Oblique 10 pt	-Adobe-Courier-Bold-O-Normal--10-100-75-75-M-60-ISO8859-1
Courier Oblique 10 pt	-Adobe-Courier-Medium-O-Normal--10-100-75-75-M-60-ISO8859-1
100 dpi Fonts	
Symbol 8 pt	-Adobe-Symbol-Medium-R-Normal--11-80-100-100-P-61-Adobe-FONTSPECIFIC
Symbol 10 pt	-Adobe-Symbol-Medium-R-Normal--14-100-100-100-P-85-Adobe-FONTSPECIFIC
Symbol 12 pt	-Adobe-Symbol-Medium-R-Normal--17-120-100-100-P-95-Adobe-FONTSPECIFIC
Symbol 14 pt	-Adobe-Symbol-Medium-R-Normal--20-140-100-100-P-107-Adobe-FONTSPECIFIC
Symbol 18 pt	-Adobe-Symbol-Medium-R-Normal--25-180-100-100-P-142-Adobe-FONTSPECIFIC
Symbol 24 pt	-Adobe-Symbol-Medium-R-Normal--34-240-100-100-P-191-Adobe-FONTSPECIFIC
Times Bold 10 pt	-Adobe-Times-Bold-R-Normal--14-100-100-100-P-76-ISO8859-1
Times Bold Italic 10 pt	-Adobe-Times-Bold-I-Normal--14-100-100-100-P-77-ISO8859-1
Times Italic 10 pt	-Adobe-Times-Medium-I-Normal--14-100-100-100-P-73-ISO8859-1
Times Roman 10 pt	-Adobe-Times-Medium-R-Normal--14-100-100-100-P-74-ISO8859-1

Chapter 11

FontProperties

All font properties are optional but will generally include the font name fields and, on a font-by-font basis, any other useful font descriptive and use information that may be required to use the font intelligently. The XLFD specifies an extensive set of standard X font properties, their interpretation, and fallback rules when the property is not defined for a given font. The goal is to provide client applications with enough font information to be able to make automatic formatting and display decisions with good typographic results.

Additional standard X font property definitions may be defined in the future and private properties may exist in X fonts at any time. Private font properties should be defined to conform to the general mechanism defined in the X Protocol to prevent overlap of name space and ambiguous property names; that is, private font property names are of the form ISO8859-1 UNDERSCORE (Column/Row 05/15), followed by the organisational identifier, followed by UNDERSCORE, and terminated with the property name.

The Backus-Naur Form syntax description of X font properties is as follows:

Properties ::=	OptFontPropList
OptFontPropList ::=	NULL | OptFontProp OptFontPropList
OptFontProp ::=	PrivateFontProp | XFontProp
PrivateFontProp ::=	STRING8 | Underscore OrganizationId Underscore STRING8
XFontProp ::=	FOUNDRY | FAMILY_NAME | WEIGHT_NAME | SLANT | SETWIDTH_NAME | ADD_STYLE_NAME | PIXEL_SIZE | POINT_SIZE | RESOLUTION_X | RESOLUTION_Y | SPACING | AVERAGE_WIDTH | CHARSET_REGISTRY | CHARSET_ENCODING | QUAD_WIDTH | RESOLUTION | MIN_SPACE | NORM_SPACE | MAX_SPACE | END_SPACE | SUPERSCRIPT_X | SUPERSCRIPT_Y | SUBSCRIPT_X | SUBSCRIPT_Y | UNDERLINE_POSITION | UNDERLINE_THICKNESS | STRIKEOUT_ASCENT | STRIKEOUT_DESCENT | ITALIC_ANGLE | X_HEIGHT | WEIGHT | FACE_NAME | COPYRIGHT | AVG_CAPITAL_WIDTH | AVG_LOWERCASE_WIDTH | RELATIVE_SETWIDTH | RELATIVE_WEIGHT | CAP_HEIGHT | SUPERSCRIPT_ SIZE | FIGURE_WIDTH | SUBSCRIPT_SIZE | SMALL_CAP_SIZE | NOTICE | DESTINATION
Underscore ::=	OCTET–the value of ISO8859-1 UNDERSCORE character (Column/Row 05/15)
OrganizationId ::=	STRING8–the registered name of the organisation

11.1 Property Definitions

11.1.1 FOUNDRY

FOUNDRY is as defined in the *FontName* except that the property type is ATOM.

FOUNDRY cannot be calculated or defaulted if not supplied as a font property.

11.1.2 FAMILY_NAME

FAMILY_NAME is as defined in the *FontName* except that the property type is ATOM.

FAMILY_NAME cannot be calculated or defaulted if not supplied as a font property.

11.1.3 WEIGHT_NAME

WEIGHT_NAME is as defined in the *FontName* except that the property type is ATOM.

WEIGHT_NAME can be defaulted if not supplied as a font property, as follows:

```
if (WEIGHT_NAME undefined) then
    WEIGHT_NAME = ATOM(''Medium'')
```

11.1.4 SLANT

SLANT is as defined in the *FontName* except that the property type is ATOM.

SLANT can be defaulted if not supplied as a font property, as follows:

```
if (SLANT undefined) then
    SLANT = ATOM(''R'')
```

11.1.5 SETWIDTH_NAME

SETWIDTH_NAME is as defined in the *FontName* except that the property type is ATOM.

SETWIDTH_NAME can be defaulted if not supplied as a font property, as follows:

```
if (SETWIDTH_NAME undefined) then
    SETWIDTH_NAME = ATOM(''Normal'')
```

11.1.6 ADD_STYLE_NAME

ADD_STYLE_NAME is as defined in the *FontName* except that the property type is ATOM.

ADD_STYLE_NAME can be defaulted if not supplied as a font property, as follows:

```
if (ADD_STYLE_NAME undefined) then
    ADD_STYLE_NAME = ATOM('''')
```

11.1.7 PIXEL_SIZE

PIXEL_SIZE is as defined in the *FontName* except that the property type is CARD32.

X clients requiring pixel values for the various typographic fixed spaces (em space, en space and thin space), can use the following algorithm for computing these values from other properties specified for a font:

```
DeciPointsPerInch = 722.7
EMspace = ROUND ((RESOLUTION_X * POINT_SIZE) / DeciPointsPerInch)
ENspace = ROUND (EMspace / 2)
THINspace = ROUND (EMspace / 3)
```

where a slash ‘‘/’’ denotes real division, the asterisk ‘‘*’’ denotes real multiplication, and ‘‘ROUND’’ denotes a function that rounds its real argument ’a’ up or down to the next integer. This rounding is done according to X = FLOOR (a + O.5), where FLOOR is a function that rounds its real argument down to the nearest integer.

PIXEL_SIZE can be approximated if not supplied as a font property, according to the following algorithm:

```
DeciPointsPerInch = 722.7
    if (PIXEL_SIZE undefined) then
    PIXEL_SIZE = ROUND ((RESOLUTION_Y * POINT_SIZE) /
        DeciPointsPerInch)
```

11.1.8 POINT_SIZE

POINT_SIZE is as defined in the *FontName* except that the property type is CARD32.

X clients requiring device-independent values for em space, en space, and thin space can use the following algorithm:

```
EMspace = ROUND (POINT_SIZE / 10)
ENspace = ROUND (POINT_SIZE / 20)
THINspace = ROUND (POINT_SIZE / 30)
```

Design POINT_SIZE cannot be calculated or approximated.

11.1.9 RESOLUTION_X

RESOLUTION_X is as defined in the *FontName* except that the property type is CARD32.

RESOLUTION_X cannot be calculated or approximated.

11.1.10 RESOLUTION_Y

RESOLUTION_Y is as defined in the *FontName* except that the property type is CARD32.

RESOLUTION_X cannot be calculated or approximated.

11.1.11 SPACING

SPACING is as defined in the *FontName* except that the property type is ATOM.

SPACING can be calculated if not supplied as a font property, according to the definitions given above for the *FontName*.

11.1.12 AVERAGE_WIDTH

AVERAGE_WIDTH is as defined in the *FontName* except that the property type is CARD32.

AVERAGE_WIDTH can be calculated if not provided as a font property, according to the following algorithm:

```
if (AVERAGE_WIDTH undefined) then
    AVERAGE_WIDTH = ROUND (MEAN (all glyph widths in font) * 10)
```

where MEAN is a function that returns the arithmetic mean of its arguments.

X clients that require values for the number of characters per inch (pitch) of a monospaced font can use the following algorithm using the AVERAGE_WIDTH and RESOLUTION_X font properties:

```
if (SPACING not proportional) then
    CharPitch = (RESOLUTION_X * 10) / AVERAGE_WIDTH
```

11.1.13 CHARSET_REGISTRY

CHARSET_REGISTRY is as defined in the *FontName* except that the property type is ATOM.

CHARSET_REGISTRY cannot be defaulted if not supplied as a font property.

11.1.14 CHARSET_ENCODING

CHARSET_ENCODING is as defined in the *FontName* except that the property type is ATOM.

CHARSET_ENCODING cannot be defaulted if not supplied as a font property.

11.1.15 MIN_SPACE

MIN_SPACE is an unsigned integer value (of type CARD32) that gives the recommended minimum word-space value to be used with this font.

MIN_SPACE can be approximated if not provided as a font property, according to the following algorithm:

```
if (MIN_SPACE undefined) then
    MIN_SPACE = ROUND(0.75 * NORM_SPACE)
```

11.1.16 NORM_SPACE

NORM_SPACE is an unsigned integer value (of type CARD32) that gives the recommended normal word-space value to be used with this font.

NORM_SPACE can be approximated if not provided as a font property, according to the following algorithm:

```
DeciPointsPerInch = 722.7
    if (NORM_SPACE undefined) then
    if (SPACE glyph exists) then
    NORM_SPACE = width of SPACE
    else NORM_SPACE = ROUND((0.33 * RESOLUTION_X * POINT_SIZE)/
        DeciPointsPerInch)
```

11.1.17 MAX_SPACE

MAX_SPACE is an unsigned integer value (of type CARD32) that gives the recommended maximum word-space value to be used with this font.

MAX_SPACE can be approximated if not provided as a font property, according to the following algorithm:

```
if (MAX_SPACE undefined) then
    MAX_SPACE = ROUND(1.5 * NORM_SPACE)
```

11.1.18 END_SPACE

END_SPACE is an unsigned integer value (of type CARD32) that gives the recommended spacing at the end of sentences.

END_SPACE can be approximated if not provided as a font property, according to the following algorithm:

```
if (END_SPACE undefined) then
    END_SPACE = NORM_SPACE
```

11.1.19 AVG_CAPITAL_WIDTH

AVG_CAPITAL_WIDTH is an integer value (of type INT32) that gives the unweighted arithmetic mean width of all the capital glyphs in the font, in tenths of pixels (applies to Latin and non-Latin fonts). For Latin fonts, capitals are the glyphs A through Z. This property is usually used for font matching or substitution.

AVG_CAPITAL_WIDTH can be calculated if not provided as a font property, according to the following algorithm:

```
if (AVG_CAPITAL_WIDTH undefined) then
    AVG_CAPITAL_WIDTH = ROUND (MEAN (capital glyph widths) * 10)
```

11.1.20 AVG_LOWERCASE_WIDTH

AVG_LOWERCASE_WIDTH is an integer value (of type INT32) that gives the unweighted arithmetic mean width of all the lower-case glyphs in the font in tenths of pixels. For Latin fonts, lower case are the glyphs a through z. This property is usually used for font matching or substitution.

Where appropriate, AVG_LOWERCASE_WIDTH can be approximated if not provided as a font property, according to the following algorithm:

```
if (AVG_LOWERCASE_WIDTH undefined) then
    if (lowercase exists) then
    AVG_LOWERCASE_WIDTH = ROUND (MEAN (lowercase glyph widths) * 10)
    else AVG_LOWERCASE_WIDTH undefined
```

11.1.21 QUAD_WIDTH

QUAD_WIDTH is an integer typographic metric (of type INT32) that gives the width of a quad (em) space.

Note: Because all typographic fixed spaces (em, en, and thin) are constant for a given font size (that is, they do not vary according to setwidth), the use of this font property has been deprecated. X clients that require typographic fixed space values are encouraged to discontinue use of QUAD_WIDTH and compute these values from other font properties (for example, PIXEL_SIZE). X clients that require a font-dependent width value should use either the FIGURE_WIDTH or one of the average character width font properties:

```
(AVERAGE_WIDTH, AVG_CAPITAL_WIDTH or AVG_LOWERCASE_WIDTH).
```

11.1.22 FIGURE_WIDTH

FIGURE_WIDTH is an integer typographic metric (of type INT32) that gives the width of the tabular figures and the dollar sign, if suitable for tabular setting (all widths equal). For Latin fonts, these tabular figures are the arabic numerals 0 through 9.

FIGURE_WIDTH can be approximated if not supplied as a font property, according to the following algorithm:

```
if (numerals and DOLLAR sign are defined & widths are equal) then
    FIGURE_WIDTH = width of DOLLAR
    else FIGURE_WIDTH property undefined
```

11.1.23 SUPERSCRIPT_X

SUPERSCRIPT_X is an integer value (of type INT32) that gives the recommended horizontal offset in pixels from the position point to the X origin of synthetic superscript text. If the current position point is at [X,Y], then superscripts should begin at [X + SUPERSCRIPT_X, Y – SUPERSCRIPT_Y].

SUPERSCRIPT_X can be approximated if not provided as a font property, according to the following algorithm:

```
if (SUPERSCRIPT_X undefined) then
    if (TANGENT(ITALIC_ANGLE) defined) then
    SUPERSCRIPT_X = ROUND((0.40 * CAP_HEIGHT) / TANGENT(ITALIC_ANGLE))
    else SUPERSCRIPT_X = ROUND(0.40 * CAP_HEIGHT)
```

where TANGENT is a trigonometric function that returns the tangent of its argument (in degrees scaled by 64).

11.1.24 SUPERSCRIPT_Y

SUPERSCRIPT_Y is an integer value (of type INT32) that gives the recommended vertical offset in pixels from the position point to the Y origin of synthetic superscript text. If the current position point is at [X,Y], then superscripts should begin at [X + SUPERSCRIPT_X, Y – SUPERSCRIPT_Y].

SUPERSCRIPT_Y can be approximated if not provided as a font property, according to the following algorithm:

```
if (SUPERSCRIPT_Y undefined) then
    SUPERSCRIPT_Y = ROUND(0.40 * CAP_HEIGHT)
```

11.1.25 SUBSCRIPT_X

SUBSCRIPT_X is an integer value (of type INT32) that gives the recommended horizontal offset in pixels from the position point to the X origin of synthetic subscript text. If the current position point is at [X,Y], then subscripts should begin at [X + SUBSCRIPT_X, Y + SUBSCRIPT_Y].

SUBSCRIPT_X can be approximated if not provided as a font property, according to the following algorithm:

```
if (SUBSCRIPT_X undefined) then
    if (TANGENT(ITALIC_ANGLE) defined) then
    SUBSCRIPT_X = ROUND((0.40 * CAP_HEIGHT) / TANGENT(ITALIC_ANGLE))
    else SUBSCRIPT_X = ROUND(0.40 * CAP_HEIGHT)
```

11.1.26 SUBSCRIPT_Y

SUBSCRIPT_Y is an integer value (of type INT32) that gives the recommended vertical offset in pixels from the position point to the Y origin of synthetic subscript text. If the current position point is at [X,Y], then subscripts should begin at [X + SUBSCRIPT_X, Y + SUBSCRIPT_Y].

SUBSCRIPT_Y can be approximated if not provided as a font property, according to the following algorithm:

```
if (SUBSCRIPT_Y undefined) then
    SUBSCRIPT_Y = ROUND(0.40 * CAP_HEIGHT)
```

11.1.27 SUPERSCRIPT_SIZE

SUPERSCRIPT_SIZE is an unsigned integer value (of type CARD32) that gives the recommended body size of synthetic superscripts to be used with this font, in pixels. This will generally be smaller than the size of the current font; that is, superscripts are imaged from a smaller font offset according to SUPERSCRIPT_X and SUPERSCRIPT_Y.

SUPERSCRIPT_SIZE can be approximated if not provided as a font property, according to the following algorithm:

```
if (SUPERSCRIPT_SIZE undefined) then
    SUPERSCRIPT_SIZE = ROUND(0.60 * PIXEL_SIZE)
```

11.1.28 SUBSCRIPT_SIZE

SUBSCRIPT_SIZE is an unsigned integer value (of type CARD32) that gives the recommended body size of synthetic subscripts to be used with this font, in pixels. As with SUPERSCRIPT_SIZE, this will generally be smaller than the size of the current font; that is, subscripts are imaged from a smaller font offset according to SUBSCRIPT_X and SUBSCRIPT_Y.

SUBSCRIPT_SIZE can be approximated if not provided as a font property, according to the algorithm:

```
if (SUBSCRIPT_SIZE undefined) then
    SUBSCRIPT_SIZE = ROUND(0.60 * PIXEL_SIZE)
```

11.1.29 SMALL_CAP_SIZE

SMALL_CAP_SIZE is an unsigned integer value (of type CARD32) that gives the recommended body size of synthetic small capitals to be used with this font, in pixels. Small capitals are generally imaged from a smaller font of slightly more weight. No offset [X,Y] is necessary.

SMALL_CAP_SIZE can be approximated if not provided as a font property, according to the following algorithm:

```
if (SMALL_CAP_SIZE undefined) then
    SMALL_CAP_SIZE = ROUND(PIXEL_SIZE * ((X_HEIGHT
    + ((CAP_HEIGHT – X_HEIGHT) / 3)) / CAP_HEIGHT))
```

11.1.30 UNDERLINE_POSITION

UNDERLINE_POSITION is an unsigned integer value (of type CARD32) that gives the recommended vertical offset in pixels from the baseline to the top of the underline. If the current position point is at [X,Y], the top of the baseline is given by [X, Y + UNDERLINE_POSITION].

UNDERLINE_POSITION can be approximated if not provided as a font property, according to the following algorithm:

```
if (UNDERLINE_POSITION undefined) then
    UNDERLINE_POSITION = ROUND((maximum descent) / 2)
```

where maximum descent is the maximum descent (below the baseline) in pixels of any glyph in the font.

11.1.31 UNDERLINE_THICKNESS

UNDERLINE_POSITION is an unsigned integer value (of type CARD32) that gives the recommended underline thickness, in pixels.

UNDERLINE_THICKNESS can be approximated if not provided as a font property, according to the following algorithm:

```
CapStemWidth = average width of the stems of capitals
    if (UNDERLINE_THICKNESS undefined) then
    UNDERLINE_THICKNESS = CapStemWidth
```

11.1.32 STRIKEOUT_ASCENT

STRIKEOUT_ASCENT is an integer value (of type INT32) that gives the vertical ascent for boxing or voiding glyphs in this font. If the current position is at [X,Y] and the string extent is EXTENT, the upper-left corner of the strikeout box is at [X, Y – STRIKEOUT_ASCENT] and the lower-right corner of the box is at [X + EXTENT, Y + STRIKEOUT_DESCENT].

STRIKEOUT_ASCENT can be approximated if not provided as a font property, according to the following algorithm:

```
if (STRIKEOUT_ASCENT undefined)
    STRIKEOUT_ASCENT =  maximum ascent
```

where maximum ascent is the maximum ascent (above the baseline) in pixels of any glyph in the font.

11.1.33 STRIKEOUT_DESCENT

STRIKEOUT_DESCENT is an integer value (of type INT32) that gives the vertical descent for boxing or voiding glyphs in this font. If the current position is at [X,Y] and the string extent is EXTENT, the upper-left corner of the strikeout box is at [X, Y – STRIKEOUT_ASCENT] and the lower-right corner of the box is at [X + EXTENT, Y + STRIKEOUT_DESCENT].

STRIKEOUT_DESCENT can be approximated if not provided as a font property, according to the following algorithm:

```
if (STRIKEOUT_DESCENT undefined)
    STRIKEOUT_DESCENT =  maximum descent
```

where maximum descent is the maximum descent (below the baseline) in pixels of any glyph in the font.

11.1.34 ITALIC_ANGLE

ITALIC_ANGLE is an integer value (of type INT32) that gives the nominal posture angle of the typeface design, in 1/64 degrees, measured from the glyph origin counterclockwise from the three o'clock position.

ITALIC_ANGLE can be defaulted if not provided as a font property, according to the following algorithm:

```
if (ITALIC_ANGLE undefined) then
    ITALIC_ANGLE = (90 * 64)
```

11.1.35 CAP_HEIGHT

CAP_HEIGHT is an unsigned integer value (of type CARD32) that gives the nominal height of the capital letters contained in the font, as specified by the FOUNDRY or typeface designer. Where applicable, it is defined to be the height of the Latin upper-case letter *X*.

Certain clients require CAP_HEIGHT to compute scale factors and positioning offsets for synthesised glyphs where this information or designed glyphs are not explicitly provided by the font (for example, small capitals, superiors, inferiors, and so on). CAP_HEIGHT is also a critical factor in font matching and substitution.

CAP_HEIGHT can be approximated if not provided as a font property, according to the following algorithm:

```
if (CAP_HEIGHT undefined) then
    if (latin font) then
        CAP_HEIGHT = XCharStruct.ascent[glyph X]
    else if (capitals exist) then
        CAP_HEIGHT = XCharStruct.ascent[some capital glyph]
    else CAP_HEIGHT undefined
```

11.1.36 X_HEIGHT

X_HEIGHT is a unsigned integer value (of type CARD32) that gives the nominal height above the baseline of the lower-case glyphs contained in the font, as specified by the FOUNDRY or typeface designer. Where applicable, it is defined to be the height of the Latin lower-case letter *x*.

As with CAP_HEIGHT, X_HEIGHT is required by certain clients to compute scale factors for synthesised small capitals where this information is not explicitly provided by the font resource. X_HEIGHT is a critical factor in font matching and substitution.

X_HEIGHT can be approximated if not provided as a font property, according to the following algorithm:

```
if (X_HEIGHT undefined) then
    if (latin font) then
        X_HEIGHT = XCharStruct.ascent[glyph x]
    else if (lowercase exists) then
        X_HEIGHT = XCharStruct.ascent[some lowercase glyph]
    else X_HEIGHT is undefined
```

11.1.37 RELATIVE_SETWIDTH

RELATIVE_SETWIDTH is an unsigned integer value (of type CARD32) that gives the coded proportionate width of the font, relative to all known fonts of the same typeface family, according to the type designer's or FOUNDRY's judgement.

The possible values are:

Code	English Translation	Description
0	Undefined	Undefined or unknown
10	UltraCondensed	The lowest ratio of average width to height.
20	ExtraCondensed	
30	Condensed	Condensed, Narrow, Compressed, ...
40	SemiCondensed	
50	Medium	Medium, Normal, Regular, ...
60	SemiExpanded	SemiExpanded, DemiExpanded, ...
70	Expanded	
80	ExtraExpanded	ExtraExpanded, Wide, ...
90	UltraExpanded	The highest ratio of average width to height.

RELATIVE_SETWIDTH can be defaulted if not provided as a font property, according to the following algorithm:

```
if (RELATIVE_SETWIDTH undefined) then
    RELATIVE_SETWIDTH = 50
```

X clients that want to obtain a calculated proportionate width of the font (that is, a font-independent way of identifying the proportionate width across all fonts and all font vendors) can use the following algorithm:

```
SETWIDTH = AVG_CAPITAL_WIDTH / (CAP_HEIGHT * 10)
```

where SETWIDTH is a real number with zero being the narrowest calculated setwidth.

11.1.38 RELATIVE_WEIGHT

RELATIVE_WEIGHT is an unsigned integer value (of type CARD32) that gives the coded weight of the font, relative to all known fonts of the same typeface family, according to the type designer's or FOUNDRY's judgement.

The possible values are:

Code	English Translation	Description
0	Undefined	Undefined or unknown
10	UltraLight	The lowest ratio of stem width to height.
20	ExtraLight	
30	Light	
40	SemiLight	SemiLight, Book, ...
50	Medium	Medium, Normal, Regular,...
60	SemiBold	SemiBold, DemiBold, ...
70	Bold	
80	ExtraBold	ExtraBold, Heavy, ...
90	UltraBold	UltraBold, Black, ..., the highest ratio of stem width to height.

RELATIVE_WEIGHT can be defaulted if not provided as a font property, according to the following algorithm:

```
if (RELATIVE_WEIGHT undefined) then
    RELATIVE_WEIGHT = 50
```

11.1.39 WEIGHT

Calculated WEIGHT is an unsigned integer value (of type CARD32) that gives the calculated weight of the font, computed as the ratio of capital stem width to CAP_HEIGHT, in the range 0 to 1000, where 0 is the lightest weight.

WEIGHT can be calculated if not supplied as a font property, according to the following algorithm:

```
CapStemWidth = average width of the stems of capitals
    if (WEIGHT undefined) then
    WEIGHT = ROUND ((CapStemWidth * 1000) / CAP_HEIGHT)
```

A calculated value for weight is necessary when matching fonts from different families because both the RELATIVE_WEIGHT and the WEIGHT_NAME are assigned by the typeface supplier, according to its tradition and practice, and therefore, are somewhat subjective. Calculated WEIGHT provides a font-independent way of identifying the weight across all fonts and all font vendors.

11.1.40 RESOLUTION

RESOLUTION is an integer value (of type INT32) that gives the resolution for which this font was created, measured in 1/100 pixels per point.

Note: As independent horizontal and vertical design resolution components are required to accommodate displays with non-square aspect ratios, the use of this font property has been deprecated, and independent RESOLUTION_X and RESOLUTION_Y font name fields/properties have been defined. X clients are encouraged to discontinue use of the RESOLUTION property and are encouraged to use the appropriate X,Y resolution properties, as required.

11.1.41 FACE_NAME

FACE_NAME is a human-understandable string (of type ATOM) that gives the full device-independent typeface name, including the owner, weight, slant, set, and so on but not the resolution, size, and so on. This property may be used as feedback during font selection.

FACE_NAME cannot be calculated or approximated if not provided as a font property.

11.1.42 COPYRIGHT

COPYRIGHT is a human-understandable string (of type ATOM) that gives the copyright information of the legal owner of the digital font data.

This information is a required component of a font but is independent of the particular format used to represent it (that is, it cannot be captured as a comment that could later be ‘‘thrown away’’ for efficiency reasons).

COPYRIGHT cannot be calculated or approximated if not provided as a font property.

11.1.43 NOTICE

NOTICE is a human-understandable string (of type ATOM) that gives the copyright information of the legal owner of the font design or, if not applicable, the trademark information for the typeface FAMILY_NAME.

Typeface design and trademark protection laws vary from country to country, the USA having no design copyright protection currently while various countries in Europe offer both design and typeface family name trademark protection. As with COPYRIGHT, this information is a required component of a font but is independent of the particular format used to represent it.

NOTICE cannot be calculated or approximated if not provided as a font property.

11.1.44 DESTINATION

DESTINATION is an unsigned integer code (of type CARD32) that gives the font design destination, that is, whether it was designed as a screen proofing font to match printer font glyph widths (WYSIWYG), as an optimal video font (possibly with corresponding printer font) for extended screen viewing (video text), and so on.

The font design considerations are very different, and at current display resolutions, the readability and legibility of these two kinds of screen fonts are very different. DESTINATION allows publishing clients that use X to model the printed page and video text clients, such as on-line documentation browsers, to query for X screen fonts that suit their particular requirements.

The encoding is as follows:

Code	English Translation	Description
0	WYSIWYG	The font is optimised to match the typographic design and metrics of an equivalent printer font.
1	Video text	The font is optimised for screen legibility and readability.

11.2 Built-in Font Property Atoms

The following font property atom definitions were predefined in the initial version of the core protocol:

Font Property/Atom Name	Property Type
MIN_SPACE	CARD32
NORM_SPACE	CARD32
MAX_SPACE	CARD32
END_SPACE	CARD32
SUPERSCRIPT_X	INT32
SUPERSCRIPT_Y	INT32
SUBSCRIPT_X	INT32
SUBSCRIPT_Y	INT32
UNDERLINE_POSITION	INT32
UNDERLINE_THICKNESS	CARD32
STRIKEOUT_ASCENT	INT32
STRIKEOUT_DESCENT	INT32
FONT_ASCENT	INT32
FONT_DESCENT	INT32
ITALIC_ANGLE	INT32
X_HEIGHT	INT32
QUAD_WIDTH	INT32 – deprecated
WEIGHT	CARD32
POINT_SIZE	CARD32
RESOLUTION	CARD32 – deprecated
COPYRIGHT	ATOM
FULL_NAME	ATOM
FAMILY_NAME	ATOM
DEFAULT_CHAR	CARD32

11.3 Scalable Fonts

The XLFD is designed to support scalable fonts. A scalable font is a font source from which instances of arbitrary size can be derived. A scalable font source might be one or more outlines together with zero or more hand-tuned bitmap fonts at specific sizes and resolutions, or it might be a programmatic description together with zero or more bitmap fonts, or some other format (perhaps even just a single bitmap font).

The following definitions are useful for discussing scalable fonts:

Well-formed XLFD pattern
: A pattern string containing 14 hyphens, one of which is the first character of the pattern. Wildcard characters are permitted in the fields of a well-formed XLFD pattern.

Scalable font name
: A well-formed XLFD pattern containing no wildcards and containing the digit ‘‘0’’ in the PIXEL_SIZE, POINT_SIZE and AVERAGE_WIDTH fields.

Scalable fields
: The XLFD fields PIXEL_SIZE, POINT_SIZE, RESOLUTION_X, RESOLUTION_Y and AVERAGE_WIDTH.

Derived instance
: The result of replacing the scalable fields of a font name with values to yield a font name that could actually be produced from the font source. A scaling engine is permitted, but not required, to interpret the scalable fields in font names to support anamorphic scaling.

Global list
: The list of names that would be returned by an X server for a *ListFonts* protocol request on the pattern ‘‘*’’ if there were no protocol restrictions on the total number of names returned.

The global list consists of font names derived from font sources. If a single font source can support multiple character sets (specified in the CHARSET_REGISTRY and CHARSET_ENCODING fields), each such character set should be used to form a separate font name in the list. For a non-scalable font source, the simple font name for each character set is included in the global list. For a scalable font source, a scalable font name for each character set is included in the list. In addition to the scalable font name, specific derived instance names may also be included in the list. The relative order of derived instances with respect to the scalable font name is not constrained. Finally, font name aliases may also be included in the list. The relative order of aliases with respect to the ‘‘real’’ font name is not constrained.

The values of the RESOLUTION_X and RESOLUTION_Y fields of a scalable font name are implementation-dependent, but to maximise backward compatibility they should be reasonable non-zero values; for example, a resolution close to that provided by the screen (in a single-screen server). Since some existing applications rely on seeing a collection of point and pixel sizes, server vendors are strongly encouraged in the near term to provide a mechanism for including, for each scalable font name, a set of specific derived instance names. For font sources that contain a collection of hand-tuned bitmap fonts, including names of these instances in the global list is recommended and sufficient.

The X Protocol request *OpenFont* on a scalable font name returns a font corresponding to an implementation-dependent derived instance of that font name.

The X Protocol request *ListFonts* on a well-formed XLFD pattern returns the following. Start with the global list. If the actual pattern argument has values containing no wildcards in scalable fields, then substitute each such field into the corresponding field in each scalable font name in the list. For each resulting font name, if the remaining scalable fields cannot be replaced with values to produce a derived instance, remove the font name from the list. Now take the

modified list, and perform a simple pattern match against the pattern argument. *ListFonts* returns the resulting list.

For example, given the global list:

```
-Linotype-Times-Bold-I-Normal--0-0-100-100-P-0-ISO8859-1
-Linotype-Times-Bold-R-Normal--0-0-100-100-P-0-ISO8859-1
-Linotype-Times-Medium-I-Normal--0-0-100-100-P-0-ISO8859-1
-Linotype-Times-Medium-R-Normal--0-0-100-100-P-0-ISO8859-1
```

a *ListFonts* request with the pattern:

```
-*-Times-*-R-Normal--*-120-100-100-P-*-ISO8859-1
```

would return:

```
-Linotype-Times-Bold-R-Normal--0-120-100-100-P-0-ISO8859-1
-Linotype-Times-Medium-R-Normal--0-120-100-100-P-0-ISO8859-1
```

ListFonts on a pattern containing wildcards that is not a well-formed XLFD pattern is only required to return the list obtained by performing a simple pattern match against the global list. X servers are permitted, but not required, to use a more sophisticated matching algorithm.

Chapter 12

Implications

12.1 Affected Elements of Xlib and the X Protocol

The following X Protocol requests must support the XLFD conventions:

- *OpenFont* – for the name argument
- *ListFonts* – for the pattern argument
- *ListFontsWithInfo* – for the pattern argument.

In addition, the following Xlib functions must support the XLFD conventions:

- *XLoadFont* – for the name argument
- *XListFontsWithInfo* – for the pattern argument
- *XLoadQueryFont* – for the name argument
- *XListFonts* – for the pattern argument.

12.2 BDF Conformance

The bitmap font distribution and interchange format adopted by the X Consortium (BDF V2.1) provides a general mechanism for identifying the font name of an X font and a variable list of font properties, but it does not mandate the syntax or semantics of the font name or the semantics of the font properties that might be provided in a BDF font. This section identifies the requirements for BDF fonts that conform to XLFD.

12.2.1 XLFD Conformance Requirements

A BDF font conforms to the XLFD specification if and only if the following conditions are satisfied:

- The value for the BDF item *FONT* conforms to the syntax and semantic definition of a XLFD *FontName* string.
- The *FontName* begins with the X *FontNameRegistry* prefix: ‘‘-’’.
- All XLFD *FontName* fields are defined.
- Any FontProperties provided conform in name and semantics to the XLFD *FontProperty* definitions.

A simple method of testing for conformance would entail verifying that the *FontNameRegistry* prefix is the string ‘‘-’’, that the number of field delimiters in the string and coded field values are valid, and that each font property name either matches a standard XLFD property name or follows the definition of a private property.

12.2.2 FONT_ASCENT, FONT_DESCENT and DEFAULT_CHAR

FONT_ASCENT, FONT_DESCENT and DEFAULT_CHAR are provided in the BDF specification as properties that are moved to the *XFontStruct* by the BDF font compiler in generating the X server-specific binary font encoding. If present, these properties shall comply with the following semantic definitions.

12.2.2.1 FONT_ASCENT

FONT_ASCENT is an integer value (of type INT32) that gives the recommended typographic ascent above the baseline for determining interline spacing. Specific glyphs of the font may extend beyond this. If the current position point for line *n* is at [X,Y], then the origin of the next line *n+1* (allowing for a possible font change) is [X, Y + $FONT\_DESCENT_n$ + $FONT\_ASCENT_{n+1}$].

FONT_ASCENT can be approximated if not provided as a font property, according to the following algorithm:

```
if (FONT_ASCENT undefined) then
    FONT_ASCENT = maximum ascent
```

where maximum ascent is the maximum ascent (above the baseline) in pixels of any glyph in the font.

12.2.2.2 FONT_DESCENT

FONT_DESCENT is an integer value (of type INT32) that gives the recommended typographic descent below the baseline for determining interline spacing. Specific glyphs of the font may extend beyond this. If the current position point for line *n* is at [X,Y], then the origin of the next line *n+1* (allowing for a possible font change) is [X, Y + $FONT\_DESCENT_n$ + $FONT\_ASCENT_{n+1}$]. The logical extent of the font is inclusive between the Y-coordinate values: Y – FONT_ASCENT and Y + FONT_DESCENT + 1.

FONT_DESCENT can be approximated if not provided as a font property, according to the following algorithm:

```
if (FONT_DESCENT undefined) then
    FONT_DESCENT = maximum descent
```

where maximum descent is the maximum descent (below the baseline) in pixels of any glyph in the font.

12.2.2.3 DEFAULT_CHAR

The DEFAULT_CHAR is a unsigned integer value (of type CARD32) that specifies the index of the default character to be used by the X server when an attempt is made to display an undefined or nonexistent character in the font. (For a font using two byte matrix format, the index bytes are encoded in the integer as byte1 * 65536 + byte2.) If the DEFAULT_CHAR itself specifies an undefined or nonexistent character in the font, then no display is performed.

DEFAULT_CHAR cannot be approximated if not provided as a font property.

Part 3:
Compound Text

X/Open Company Ltd.

Chapter 13

Compound Text

Compound Text is a format for multiple character set data, such as multi-lingual text. The format is based on ISO standards for encoding and combining character sets. Compound Text is intended to be used in three main contexts: inter-client communication using selections (as defined in the **ICCCM** specification); window properties (for example, window manager hints as defined in the **ICCCM** specification); and resources (for example, as defined in the **Xlib - C Language Binding** specification and **Toolkit Intrinsics** specification.

Compound Text is intended as an external representation, or interchange format, not as an internal representation. It is expected (but not required) that clients will convert Compound Text to some internal representation for processing and rendering, and convert from that internal representation to Compound Text when providing textual data to another client.

13.1 Status

This specification is syntactically and semantically complete.

All the facilities specified in this specification are mandatory.

There are no parts of this specification which are optional.

Internationalisation

The X Window System is 8-bit transparent. Any 8-bit or 16-bit codeset may be used in the font and text calls. In addition, 8-bit codesets may be used in all strings including filenames, atom names and color names.

13.2 Values

The name of this encoding is ‘‘COMPOUND_TEXT’’. When text values are used in the ICCCM-compliant selection mechanism or are stored as window properties in the server, the type used should be the atom for ‘‘COMPOUND_TEXT’’.

Octet values are represented in this document as two decimal numbers in the form col/row. This means the value (col * 16) + row. For example, 02/01 means the value 33.

For our purposes, the octet encoding space is divided into four ranges:

C0 octets from 00/00 to 01/15

GL octets from 02/00 to 07/15

C1 octets from 08/00 to 09/15

GR octets from 10/00 to 15/15

C0 and C1 are ‘‘control character’’ sets, while GL and GR are ‘‘graphic character’’ sets. Only a subset of C0 and C1 octets are used in the encoding, and depending on the character set encoding defined as GL or GR, a subset of GL and GR octets may be used; see below for details. All octets (00/00 to 15/15) may appear inside the text of extended segments (defined below).[13]

13. For those familiar with ISO 2022, we will use only an 8-bit environment, and we will always use G0 for GL and G1 for GR.

13.3 Control Characters

In C0, only the following values will be used:

00/09	HT	HORIZONTAL TABULATION
00/10	NL	NEW LINE
01/11	ESC	(ESCAPE)

In C1, only the following value will be used:[14]

09/11	CSI	CONTROL SEQUENCE INTRODUCER

No control sequences are defined in Compound Text for changing the C0 and C1 sets.

A horizontal tab can be represented with the octet 00/09. Specification of tabulation width settings is not part of Compound Text, and must be obtained from context (in an unspecified manner).[15]

A newline (line separator/terminator) can be represented with the octet 00/10.[16]

The remaining C0 and C1 values (01/11 and 09/11) are only used in the control sequences defined below.

14. The alternate 7-bit CSI encoding 01/11 05/11 is not used in Compound Text.

15. Inclusion of horizontal tab is for consistency with the STRING type currently defined in the **ICCCM** specification.

16. Note that 00/10 is normally LINEFEED, but is being interpreted as NEWLINE. This can be thought of as using the (deprecated) NEW LINE mode, E.1.3, in ISO/IEC 6429. Use of this value instead of 08/05 (NEL, NEXT LINE) is for consistency with the STRING type currently defined in the **ICCCM** specification.

13.4 Standard Character Set Encodings

The default GL and GR sets in Compound Text correspond to the left and right halves of ISO 8859-1 (Latin 1). As such, any legal instance of a STRING type (as defined in the **ICCCM** specification) is also a legal instance of type COMPOUND_TEXT.[17]

To define one of the approved standard character set encodings to be the GL set, one of the following control sequences is used:

01/11 02/08 {I} F	94 character set
01/11 02/04 02/08 {I} F	94^N character set

To define one of the approved standard character set encodings to be the GR set, one of the following control sequences is used:

01/11 02/09 {I} F!94 character set
01/11 02/13 {I} F!96 character set
01/11 02/04 02/09 {I} F!94^N character set

The ''F'' in the control sequences above stands for ''Final character'', which is always in the range 04/00 to 07/14. The ''{I}'' stands for zero or more ''intermediate characters'', which are always in the range 02/00 to 02/15, with the first intermediate character always in the range 02/01 to 02/03. The registration authority has defined an ''{I} F'' sequence for each registered character set encoding.[18]

For GL, octet 02/00 is always defined as SPACE, and octet 07/15 (normally DELETE) is never used. For a 94-character set defined as GR, octets 10/00 and 15/15 are never used.[19]

A 94^N character set uses N octets (N>1) for each character. The value of N is derived from the column value for F:

column 04 or 05	2 octets
column 06	3 octets
column 07	4 or more octets

In a 94^N encoding, the octet values 02/00 and 07/15 (in GL) and 10/00 and 15/15 (in GR) are never used.[20]

Once a GL or GR set has been defined, all further octets in that range (except within control sequences and extended segments) are interpreted with respect to that character set encoding, until the GL or GR set is redefined. GL and GR sets can be defined independently, they do not have to be defined in pairs.

Note that when actually using a character set encoding as the GR set, you must force the most significant bit (08/00) of each octet to be a one, so that it falls in the range 10/00 to 15/15.[21]

---

17. The implied initial state in ISO 2022 is defined with the sequence:

01/11 02/00 04/03	GO and G1 in an 8-bit environment only. Designation also invokes.
01/11 02/00 04/07	In an 8-bit environment, C1 represented as 8-bits.
01/11 02/00 04/09	Graphic character sets can be 94 or 96.
01/11 02/00 04/11	8-bit code is used.
01/11 02/08 04/02	Designate ASCII into G0.
01/11 02/13 04/01	Designate right-hand part of ISO Latin-1 into G1.

18. Final characters for private encodings (in the range 03/00 to 03/15) are not permitted here in Compound Text.

19. This is consistent with ISO 2022.

20. The column definitions come from ISO 2022.

21. Control sequences to specify character set encoding revisions (as in Section 6.3.13 of ISO 2022) are not used in Compound Text. Revision indicators do not appear to provide useful information in the context of Compound Text. The most recent revision can always be assumed, since revisions are upward-compatible.

13.5 Approved Standard Encodings

The following are the approved standard encodings to be used with Compound Text. Note that none have Intermediate characters; however, a good parser will still deal with Intermediate characters in the event that additional encodings are later added to this list.

{I} F	94/96	Description
04/02	94	7-bit ASCII graphics (ANSI X3.4-1968), Left half of ISO 8859 sets
04/09	94	Right half of JIS X0201-1976 (reaffirmed 1984), 8-Bit Alphanumeric-Katakana Code
04/10	94	Left half of JIS X0201-1976 (reaffirmed 1984), 8-Bit Alphanumeric-Katakana Code
04/01	96	Right half of ISO 8859-1, Latin alphabet No. 1
04/02	96	Right half of ISO 8859-2, Latin alphabet No. 2
04/03	96	Right half of ISO 8859-3, Latin alphabet No. 3
04/04	96	Right half of ISO 8859-4, Latin alphabet No. 4
04/06	96	Right half of ISO 8859-7, Latin/Greek alphabet
04/07	96	Right half of ISO 8859-6, Latin/Arabic alphabet
04/08	96	Right half of ISO 8859-8, Latin/Hebrew alphabet
04/12	96	Right half of ISO 8859-5, Latin/Cyrillic alphabet
04/13	96	Right half of ISO 8859-9, Latin alphabet No. 5
04/01	94^2	GB2312-1980, China (PRC) Hanzi
04/02	94^2	JIS X0208-1983, Japanese Graphic Character Set
04/03	94^2	KS C5601-1987, Korean Graphic Character Set

The sets listed as ‘‘Left half of ...’’ should always be defined as GL. The sets listed as ‘‘Right half of ...’’ should always be defined as GR. Other sets can be defined either as GL or GR.

13.6 Non-standard Character Set Encodings

Character set encodings that are not in the list of approved standard encodings can be included using ‘‘extended segments’’. An extended segment begins with one of the following sequences:[22]

01/11 02/05 02/15 03/00 M L	variable number of octets per character
01/11 02/05 02/15 03/01 M L	1 octet per character
01/11 02/05 02/15 03/02 M L	2 octets per character
01/11 02/05 02/15 03/03 M L	3 octets per character
01/11 02/05 02/15 03/04 M L	4 octets per character

The ‘‘M’’ and ‘‘L’’ octets represent a 14-bit unsigned value giving the number of octets that appear in the remainder of the segment. The number is computed as ((M - 128) * 128) + (L - 128). The most significant bit M and L are always set to one. The remainder of the segment consists of two parts, the name of the character set encoding and the actual text. The name of the encoding comes first, and is separated from the text by the octet 00/02 (STX, START OF TEXT). Note that the length defined by M and L includes the encoding name and separator.[23]

The name of the encoding should be registered with the X Consortium to avoid conflicts, and should when appropriate match the CharSet Registry and Encoding registration used in the X Logical Font Description. The name itself should be encoded using ISO 8859-1 (Latin 1), should not use question mark (03/15) or asterisk (02/10), and should use hyphen (02/13) only in accordance with the X Logical Font Description.

Extended segments are not to be used for any character set encoding which can be constructed from a GL/GR pair of approved standard encodings. For example, it is incorrect to use an extended segment for any of the ISO 8859-1 family of encodings.

It should be noted that the contents of an extended segment are arbitrary; for example, they may contain octets in the C0 and C1 ranges, including 00/00, and octets comprising a given character may differ in their most significant bit.[24]

---

22. This uses the ‘‘other coding system’’ of ISO 2022, using private Final characters.

23. The encoding of the length is chosen to avoid having zero octets in Compound Text when possible, because embedded NUL values are problematic in many C language routines. The use of zero octets cannot be ruled out entirely however, since some octets in the actual text of the extended segment may have to be zero.

24. ISO registered ‘‘other coding systems’’ are not used in Compound Text; extended segments are the only mechanism for non-ISO 2022 encodings.

13.7 Directionality

If desired, horizontal text direction can be indicated using the following control sequences:[25]

09/11 03/01 05/13	begin left-to-right text
09/11 03/02 05/13	begin right-to-left text
09/11 05/13	end of string

Directionality can be nested. Logically, a stack of directions is maintained. Each of the first two control sequences pushes a new direction on the stack, and the third sequence (revert) pops a direction from the stack. The stack starts out empty at the beginning of a Compound Text string. When the stack is empty, the directionality of the text is unspecified.

Directionality applies to all subsequent text, whether in GL, GR or an extended segment. If the desired directionality of GL, GR, or extended segments differ, then directionality control sequences must be inserted when switching between them.

Note that definition of GL and GR sets is independent of directionality; defining a new GL or GR set does not change the current directionality, and pushing or popping a directionality does not change the current GL and GR definitions.

Specification of directionality is entirely optional; text direction should be clear from context in most cases. However, it must be the case that either all characters in a Compound Text string have explicitly specified direction, or that all characters have unspecified direction. That is, if directionality control sequences are used, the first such control sequence must precede the first graphic character in a Compound Text string, and graphic characters are not permitted whenever the directionality stack is empty.

25. This is a subset of the SDS (START DIRECTED STRING) control in the Draft Bidirectional Addendum to ISO/IEC 6429.

13.8 Resources

To use Compound Text in a resource, you can simply treat all octets as if they were ASCII/Latin-1, and just replace all ‘‘\’’ octets (05/12) with the two octets ‘‘\\’’, all newline octets (00/10) with the two octets ‘‘\n’’, and all zero octets with the four octets ‘‘\000’’. It is up to the client making use of the resource to interpret the data as Compound Text; the policy by which this is ascertained is not constrained by the **Compound Text** specification.

13.9 Font Names

The following CharSet names for the standard character set encodings are registered for use in font names under the X Logical Font Description:

Name	Encoding Standard
ISO8859-1	ISO 8859-1
ISO8859-2	ISO 8859-2
ISO8859-3	ISO 8859-3
ISO8859-4	ISO 8859-4
ISO8859-5	ISO 8859-5
ISO8859-6	ISO 8859-6
ISO8859-7	ISO 8859-7
ISO8859-8	ISO 8859-8
ISO8859-9	ISO 8859-9
JISX0201.1976-0	JIS X0201-1976 (reaffirmed 1984)
GB2312.1980-0	GB2312-1980, GL encoding
JISX0208.1983-0	JIS X0208-1983, GL encoding
KSC5601.1987-0	KS C5601-1987, GL encoding

13.10 Extensions

There is no absolute requirement for a parser to deal with anything but the particular encoding syntax defined in this specification. However, it is possible that Compound Text may be extended in the future, and as such it may be desirable to construct the parser to handle ISO 2022/ISO/IEC 6429 syntax more generally.

There are two general formats covering all control sequences that are expected to appear in extensions:

01/11 {I} F
: For this format, I is always in the range 02/00 to 02/15, and F is always in the range 03/00 to 07/14.

09/11 {P} {I} F
: For this format, P is always in the range 03/00 to 03/15, I is always in the range 02/00 to 02/15, and F is always in the range 04/00 to 07/14.

  In addition, new (singleton) control characters (in the C0 and C1 ranges) might be defined in the future.

  Finally, new kinds of ''segments'' might be defined in the future using syntax similar to extended segments.

01/11 02/05 02/15 F M L
: For this format, F is in the range 03/05 to 3/15. M and L are as defined in extended segments. Such a segment will always be followed by the number of octets defined by M and L. These octets can have arbitrary values, and need not follow the internal structure defined for current extended segments.

If extensions to this specification are defined in the future, then any string incorporating instances of such extensions must start with one of the following control sequences:

01/11 02/03 V 03/00	ignoring extensions is OK
01/11 02/03 V 03/01	ignoring extensions is not OK

In either case, V is in the range 02/00 to 02/15 and indicates the major version minus one of the specification being used. These version control sequences are for use by clients that implement earlier versions, but have implemented a general parser. The first control sequence indicates that it is acceptable to ignore all extension control sequences; no mandatory information will be lost in the process. The second control sequence indicates that it is unacceptable to ignore any extension control sequences; mandatory information would be lost in the process. In general, it will be up to the client generating the Compound Text to decide which control sequence to use.

13.11 Errors

If a Compound Text string does not match the specification here (for example, uses undefined control characters, or undefined control sequences, or incorrectly formatted extended segments), it is best to treat the entire string as invalid, except as indicated by a version control sequence.

Part 4:
Bitmap Distribution Format (BDF)

X/Open Company Ltd.

Chapter 14

Bitmap Distribution Format (BDF)

The Bitmap Distribution Format (BDF), Version 2.1, is an X Consortium standard for font interchange, intended to be easily understood by both humans and computers.

14.1 Status

This specification is syntactically and semantically complete.

X/Open-compliant X Window systems are not required to provide a mechanism for importing of externally generated fonts. However, if a system does provide such a mechanism, then it must at least provide a method whereby fonts specified in the format described in this specification can be converted mechanically to the format required for import.

Internationalisation

The X Window System is 8-bit transparent. Any 8-bit or 16-bit codeset may be used in the font and text calls. In addition, 8-bit codesets may be used in all strings including filenames, atom names and color names.

14.2 File Format

Character bitmap information will be distributed in an USASCII-encoded, human-readable form. Each file is encoded in the printable characters (octal 40 through 176) of USASCII plus carriage return and linefeed. Each file consists of a sequence of variable-length lines. Each line is terminated either by a carriage return (octal 015) and linefeed (octal 012) or by just a linefeed.

The information about a particular family and face at one size and orientation will be contained in one file. The file begins with information pertaining to the face as a whole, followed by the information and bitmaps for the individual characters.

A font bitmap description file has the following general form, where each item is contained on a separate line of text in the file. Tokens on a line are separated by spaces. Keywords are in upper-case, and must appear in upper-case in the file.

1. The word STARTFONT followed by a version number indicating the exact file format used. The version described here is 2.1.
2. Lines beginning with the word COMMENT may appear anywhere between the STARTFONT line and the ENDFONT line. These lines are ignored by font compilers.
3. The word FONT followed by either the XLFD font name (as specified in part III) or some private font name. Creators of private font name syntaxes are encouraged to register unique font name prefixes with the X Consortium to prevent naming conflicts. Note that the name continues all the way to the end of the line and may contain spaces.
4. The word SIZE followed by the *point size* of the characters, the *x resolution* and the *y resolution* of the device for which these characters were intended.
5. The word FONTBOUNDINGBOX followed by the *width in x*, *height in y*, and the x and y displacement of the lower left corner from the *origin*. (See the examples in the next section.)
6. Optionally, the word STARTPROPERTIES followed by the number of properties (*p*) that follow.
7. Then come *p* lines consisting of a word for the *property name* followed by either an integer or string surrounded by double-quote (octal 042). Internal double-quote characters are indicated by using two in a row.

   Properties named FONT_ASCENT, FONT_DESCENT and DEFAULT_CHAR should be provided to define the logical font-ascent and font-descent and the default-char for the font. These properties will be removed from the actual font properties in the binary form produced by a compiler. If these properties are not provided, a compiler may reject the font or may compute (arbitrary) values for these properties.
8. The property section, if it exists, is terminated by ENDPROPERTIES.
9. The word CHARS followed by the number of character segments (*c*) that follow.
10. Then come *c* character segments, as described in the next section.
11. The file is terminated with the word ENDFONT.

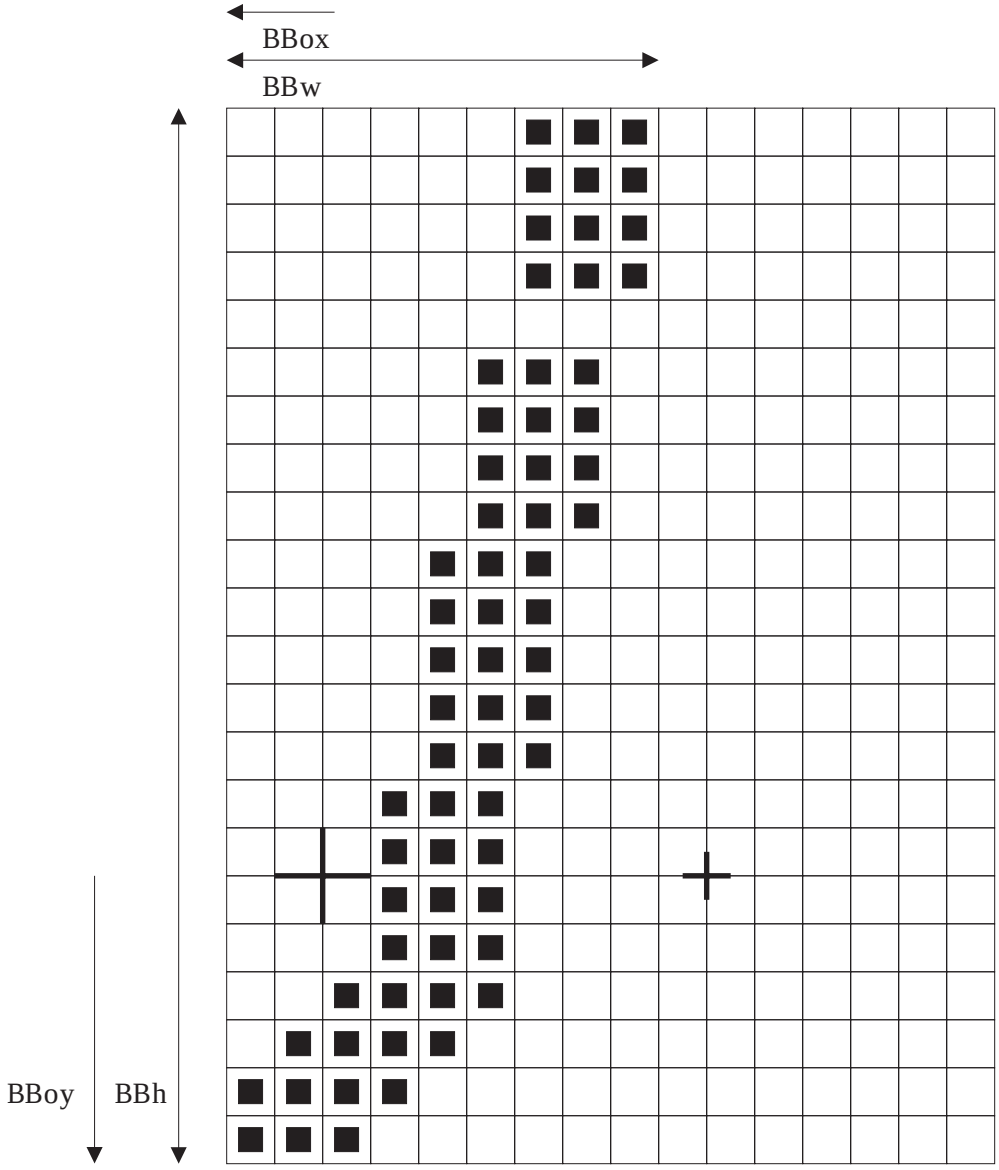
14.3 Format of Character Segments

The format of a character segment is as follows, where each list item describes one or more separate lines of text in the file. Items on a line in the files are separated by spaces.

1. The word STARTCHAR followed by up to 14 characters (no blanks) of descriptive *name* of the glyph.
2. The word ENCODING followed by one of the following forms:
   - <n> – the glyph index, that is, a positive integer representing the character code used to access the glyph in X requests, as defined by the encoded character set given by the CHARSET_REGISTRY-CHARSET_ENCODING font properties for XLFD conforming fonts. If these XLFD font properties are not defined, the encoding scheme is font-dependent.
   - –1 <n> – equivalent to form above. This syntax is provided for backward compatibility with previous versions of this specification and is not recommended for use with new fonts.
   - –1 – an unencoded glyph. Some font compilers may discard unencoded glyphs, but, in general, the glyph names may be used by font compilers and X servers to implement dynamic mapping of glyph repertoires to character encodings as seen through the X Protocol.
3. The word SWIDTH followed by the *scalable width* in x and y of character. Scalable widths are in units of 1/1000th of the size of the character. If the size of the character is *p* points, the width information must be scaled by *p*/1000 to get the width of the character in printer's points. This width information should be considered as a vector indicating the position of the next character's origin relative to the origin of this character. To convert the scalable width to the width in device pixels, multiply SWIDTH times *p*/1000 times *r*/72, where *r* is the device resolution in pixels per inch. The result is a real number giving the ideal print width in device pixels. The actual device width must of course be an integral number of device pixels and is given in the next entry. The SWIDTH y value should always be zero for a standard X font.
4. The word DWIDTH followed by the width in x and y of the character in device units. Like the SWIDTH, this width information is a vector indicating the position of the next character's origin relative to the origin of this character. Note that the DWIDTH of a given ''hand-tuned'' WYSIWYG glyph may deviate slightly from its ideal device-independent width given by SWIDTH in order to improve its typographic characteristics on a display. The DWIDTH y value should always be zero for a standard X font.
5. The word BBX followed by the width in x (*BBw*), *height* in y (*BBh*) and x and y displacement (*BBox*, *BBoy*) of the lower left corner from the *origin* of the character.
6. The optional word ATTRIBUTES followed by the attributes as 4 *hex-encoded* characters. The interpretation of these attributes is undefined in this document.
7. The word BITMAP.
8. *h* lines of *hex-encoded* bitmap, padded on the right with zeros to the nearest byte (that is, multiple of 8).
9. The word ENDCHAR.

14.4 Metric Information

Figure 14-1 and Figure 14-2 on page 125 best illustrate the bitmap format and character metric information.



BBw = 9, BBh = 22, BBox = -2, BBoy = -6
DWIDTH = 8 0 SWIDTH = 355 0
+ = character origin and width

Figure 14-1 An Example of a Descender

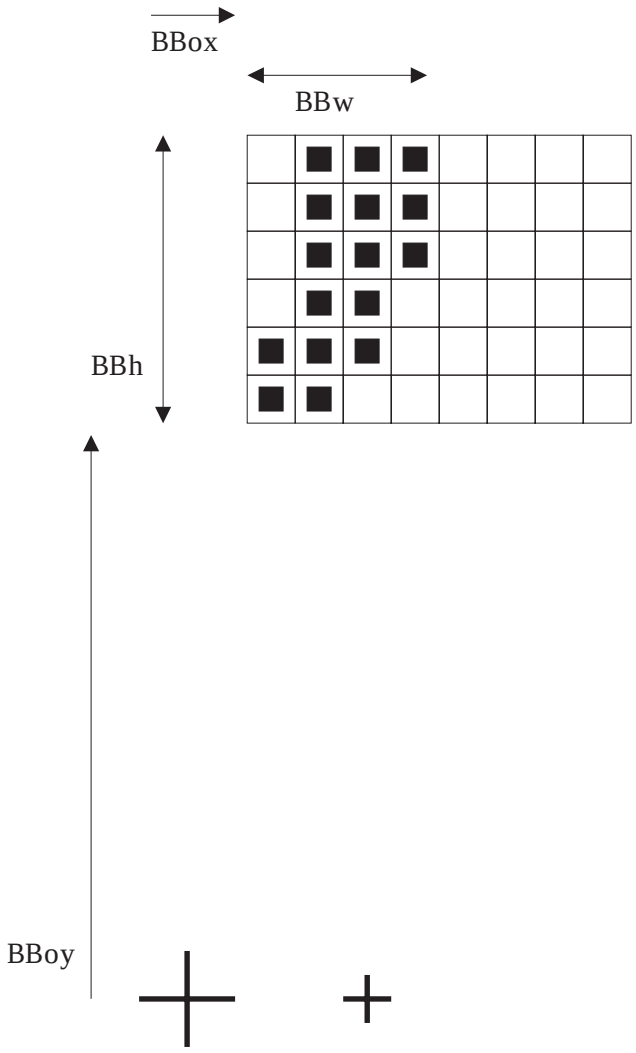


Figure 14-2 An Example with the Origin Outside the Bounding Box

14.5 An Example File

The following is an abbreviated example of a bitmap file containing the specification of two characters (the j and quoteright in Figure 14-1 on page 124 and Figure 14-2 on page 125).

```
STARTFONT 2.1
COMMENT This is a sample font in 2.1 format.
FONT -Adobe-Helvetica-Bold-R-Normal--24-240-75-75-P-65-ISO8859-1
SIZE 24 75 75
FONTBOUNDINGBOX 9 24 -2 -6
STARTPROPERTIES 19
FOUNDRY "Adobe"
FAMILY "Helvetica"
WEIGHT_NAME "Bold"
SLANT "R"
SETWIDTH_NAME "Normal"
ADD_STYLE_NAME ""
PIXEL_SIZE 24
POINT_SIZE 240
RESOLUTION_X 75
RESOLUTION_Y 75
SPACING "P"
AVERAGE_WIDTH 65
CHARSET_REGISTRY "ISO8859"
CHARSET_ENCODING "1"
MIN_SPACE 4
FONT_ASCENT 21
FONT_DESCENT 7
COPYRIGHT "Copyright (c) 1987 Adobe Systems, Inc."
NOTICE "Helvetica is a registered trademark of Linotype Inc."
ENDPROPERTIES
CHARS 2
STARTCHAR j
ENCODING 106
SWIDTH 355 0
DWIDTH 8 0
BBX 9 22 -2 -6
BITMAP
0380
0380
0380
0380
0000
0700
0700
0700
0700
0E00
0E00
0E00
0E00
0E00
1C00
```

```
1C00
1C00
1C00
3C00
7800
F000
E000
ENDCHAR
STARTCHAR quoteright
ENCODING 39
SWIDTH 223 0
DWIDTH 5 0
BBX 4 6 2 12
ATTRIBUTES 01C0
BITMAP
70
70
70
60
E0
C0
ENDCHAR
ENDFONT
```

Index