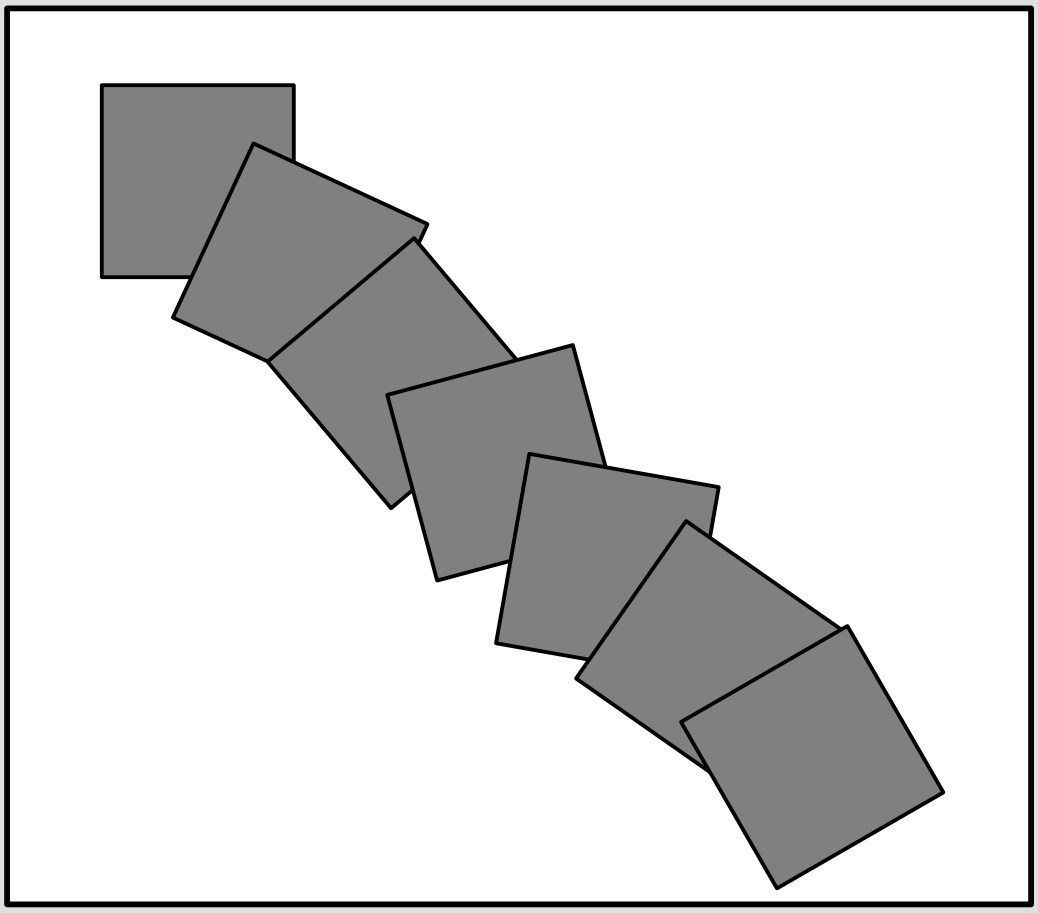
Implementing Cut and Paste in the X Environment



Notice

The information contained in this document is subject to change without notice.

HEWLETT−PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett−Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Permission to use, copy, and distribute this documentation is hereby granted, provided that the copyright notice below and this permission notice appear in supporting documentation, and that the name of Hewlett−Packard not be used in advertising or publicity pertaining to distribution of the documentation without specific, written prior permission.

September 1993. First external review release.
October 1993. First public release.

© Copyright 1993 Hewlett−Packard Company.

1 Intro 1
What and Why This Is 1
Terminology 2
Roadmap 2
Feedback 2

2 Theory 4
Outline 4
The Source of All Wisdom 4
Jargon 5
Scenario 1: A Successful Transaction 5
Scenario 2: Losing My Selection 8
PRIMARY vs. CLIPBOARD 10
CLIPBOARD Scenario 11
What's a Poor Developer To Do? 14
A Word About Cut Buffers 16

3 Practice 17
Overview 17
The Xt Selection API 17
The Sample Program 21

4 Targets 25
Rationale / Background 25
Standardization 25
Existing Standard Targets 26
The Future 30

5 The selectinfo Program 31
Background / Rationale 31
Program Overview 31
Using selectinfo 32

A The Sample Program 34

B Applications 46
Interleaf 5.3.0 46
Lotus 1-2-3 Version 1.2 47
Lucid Emacs 47
R5 xterm 47
Motif 1.1 Text Widget 48
Motif 1.2 Text Widget 48
R5 Athena Text Widget 48

1 Intro

What and Why This Is

This document and accompanying programs are intended to help X11 application developers better support data communication with other applications via cut and paste. For reasons that aren't entirely clear to me, many existing X applications, both commercial and freely distributed, do not adequately support cut and paste. At best, applications can exchange ASCII data; at worst, they implement cut and paste incorrectly, or don't implement it at all.

Why is this? The X cut and paste protocols are well-defined by the InterClient Communications Conventions Manual (ICCCM). The Xt Intrinsics provide a straightforward API that implements that protocol, hiding most of the nasty details from the application programmer. What possible explanations are there for applications not making use of this API?

I can see at least two areas of confusion:

- Lack of consensus on formats for non-textual data.
- Absence of widely distributed source for a sample program that implements cut and paste correctly. [1]

Further, this is a chicken and egg problem. It may be difficult for a developer to see any payback from the effort required when none of the other applications out there will communicate with her well-behaved program.

It is my hope that this document will help break the logjam by addressing these two areas. The first by publicizing the existing mechanism for registering data formats and by providing a reference list of applications and the formats they support for data exchange. It is obvious how to address the second point: write such a program and distribute it. The source for just such a sample program appears in Ap-

1. This is not strictly true. O'Reilly Volume IV contains source for such a program. Perhaps the real problem is that many of the most widely distributed programs (in source form) do not implement cut and paste correctly.

pendix A. An electronic version can be obtained via anonymous ftp from `ftp.x.org` in the file `contrib/select_progs.tar.Z`.

Additionally, to assist in compiling information about what applications support what data formats, I have written a simple program to display information about the current selection owner. This is distributed in the same ways as the sample program.

Terminology

I assume that you are familiar with standard X terminology (server, client, event, etc.); if you're not, this document will probably not make much sense to you.

Beyond that, the only term needing clarification is "cut and paste". In this document, "cut and paste" means a technique by which a user can easily transfer data from one application on their X display to another running on that same display. This definition is slightly broader than that used by others, but I think it is closer to what most users think of when they hear those words.

Roadmap

The remaining sections of this document are:

Theory: An overview of the mechanism underlying cut and paste in the X environment. Lists existing pertinent documents and where to obtain them.

Practice: Step by step analysis of a well-behaved cut and paste sample program that uses the API provided by the Xt Intrinsics.

Targets: A discussion of formats for data exchange. Lists the data formats in common use by X applications. Describes the procedure for registering new formats. Suggests some possible routes to standardization.

The selectinfo Program: A guide to using the selectinfo tool.

Appendix A: Source for the sample program, `display_text`.

Appendix B: List of applications with descriptions of how they support cut and paste.

Feedback

I am very interested in your reactions to this document. Did you find it useful? Did it help you to change your application to better support cut and paste? What changes might be made to improve it?

I am also counting on your assistance in maintaining the list of supported formats. If you have access to an application not listed in Appendix A or B, use the `selectinfo` program to determine what formats it supports and send me the results.

Any and all feedback should be e-mailed to me at `chan@fc.hp.com`. If you do not have e-mail access, as a last resort you can send your comments via US Mail to

Chan Benson
Hewlett-Packard Company
3404 East Harmony Road
Fort Collins, CO 80525
USA

2 Theory

Outline

This section provides the technical background required to understand the infrastructure for cut and paste in the X environment. It begins with an explanation of where the rules for the cut and paste procedure originate, then defines the terms required to discuss that procedure. Said discussion follows. Finally the section wraps up with a treatment of the two competing models for using selections and a presentation of alternatives for integrating those models.

The principles and concepts developed in this section will be made tangible in the next section, where we go over the source for the sample well-behaved application point by point.

The Source of All Wisdom

The authoritative source for the rules governing how to implement inter-application cut and paste is the ICCCM (InterClient Communications Conventions Manual, typically pronounced “eye triple see em”). This document, originally released in 1989, resulted from collaboration among software engineers from various workstation manufacturers and the X Consortium. It defines the standards for how X client programs should behave in order to cooperate with window managers and other clients in the X environment. Section 2 of the ICCCM, which describes how X clients should exchange data, is required reading for anyone writing a well-behaved X program (actually the entire document is, but at the moment I’m only concerned with the part that applies to cut and paste).

Now, the ICCCM may be authoritative, but it’s not light reading; for those not familiar with the details of the underlying issues it can be tough to figure out exactly what it’s saying. So herewith, I present my easy-listening version, intended to give an overview of the process described in the ICCCM. I am not attempting to replace the ICCCM, but to provide the information in a higher-level form in the hope of

making the legalese in the ICCCM more understandable to the novice.

Jargon

It is essential to understand the basic concepts underlying the implementation of cut and paste under X and to know the words associated with those concepts.

The mechanism for implementing cut and paste under X is based on *selections*. Purists have probably been frowning over my use of the term “cut and paste” throughout this document because we’re really talking about selections. What is a selection? I’ve never actually seen it rigorously defined anywhere, but I’ll give it my best shot: a selection is a named token controlled by the X server. An X client can request *ownership* of a particular selection. By so doing, it is volunteering to handle requests from other clients for the value of that selection.

A selection owner can lose ownership by surrendering it intentionally, or can lose it forcibly when another client asserts ownership of that selection. So at any given moment, a selection is owned by one or zero clients. Since X does not provide a way to directly name a client, the owner of a selection is identified by the ID of a window created by the owning client.

There can be as many selections as there are possible names for them, but for cut and paste, we are concerned only with the PRIMARY and CLIPBOARD selections. In fact, let’s assume for the moment that we’re talking only of the PRIMARY selection, and I’ll explain the difference between PRIMARY and CLIPBOARD at the end of this section.

The process of exchanging data via selections involves two clients: the client that is the source for the data and the client that is requesting the data. I shall refer to these clients as the *owner* and *requestor* respectively.

When a requestor asks for the value of a given selection, it must specify the format in which it would like the data delivered, and the owner (if possible) will convert the selection data to that format. The format specified in a request for the value of a selection is called a *target*. Targets, like selections, are named by X atoms.

Now that we’ve defined the required jargon we can proceed with descriptions of a few scenarios in the use of the selection mechanism.

Scenario 1: A Successful Transaction

First we describe the typical sequence of events involved when a user cuts some data from one application and pastes it into another. I do

not detail all the mechanisms used to communicate between the owner and the requestor. That level of detail can be found in the ICCCM, but you do not actually need to understand it to use the Xt API for cut and paste covered in the next section.

In our scenario, client A is the owner and client B is the requestor.

The process begins when the user makes some gesture that indicates to client A that she wishes to make some data from client A available to other clients. For instance, she may click and drag the mouse over a section of text that she wishes to copy.

At this point there are two things client A should do:

1. Assert ownership of the PRIMARY selection.
2. If step one is successful, visually highlight the selected data. For text, this typically means invert the selected range.

One of the parameters required to gain ownership of a selection is a timestamp corresponding to the time at which ownership should commence. A program should never use `CurrentTime` for this timestamp. In almost every case, the appropriate timestamp is the one from the X event corresponding to the user gesture that requested the cut or copy.

Next, the user turns her attention to client B, where she wishes to paste the selected data. She performs some pasting gesture. The natural response for client B is to request the data associated with the PRIMARY selection. But how does client B know in what format(s) the data is available? Given that X applications encompass more than just textual data, how does client B even know that any of the available formats are appropriate for pasting in the context indicated by the user? The answer is that it doesn't; the selection requestor must ask the owner for a list of formats in which the owner can provide the selection. It does this by making a request for the selection with a target of TARGETS.

Just like when a client asserts ownership of a selection, making a request for the value of a selection requires a timestamp. Once again, a program should use the timestamp from the X event corresponding to the user gesture that motivated the request.

After the requestor makes the request for TARGETS, the ball is back in the owner's court; it responds to this request with a list of atoms naming all the formats (targets) it can convert the current selection to.

Now client B (the requestor) can decide which of the supported formats (if any) is most appropriate and re-request the selection value with the desired format as the target.

The owner responds to this new selection request, this time by delivering the value of the selection in the format specified.

Finally, when the data is delivered client B copies it into the appropriate place.

A diagram may help you to better understand the sequence of events:

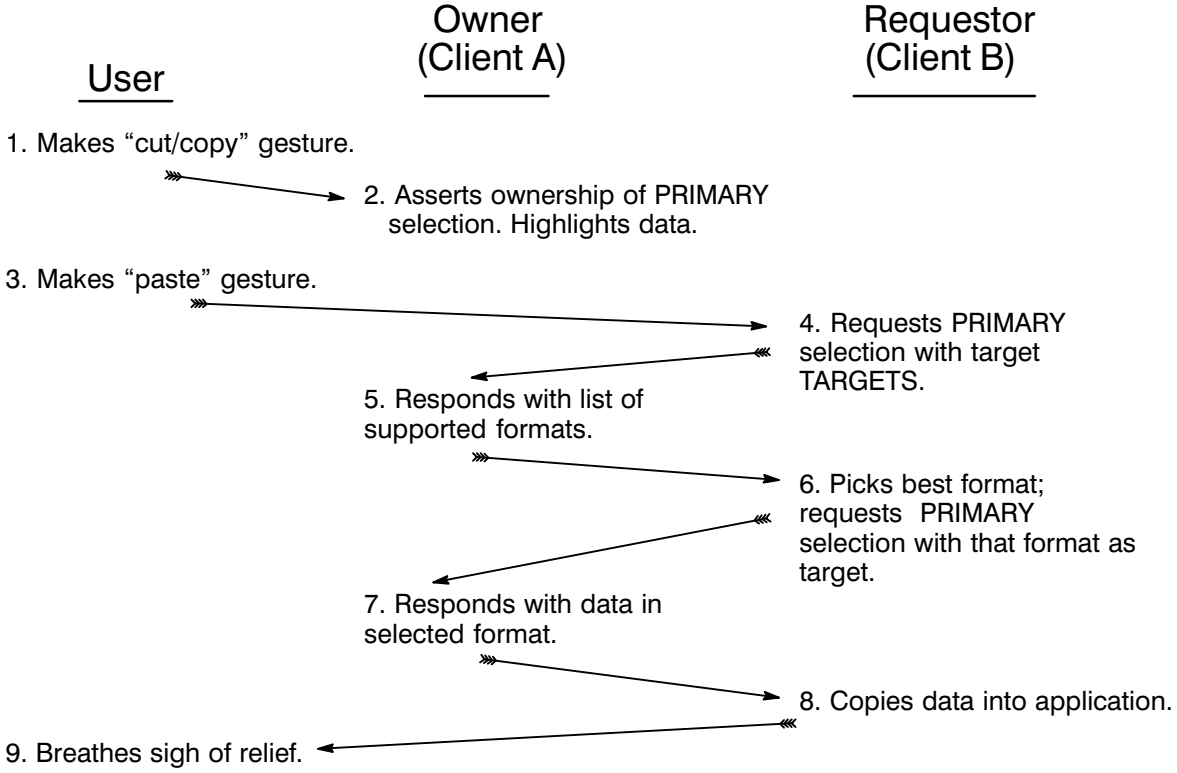


Figure 1: The process of data exchange (cut and paste) via selections

Quite a few details were omitted from that explanation. Most of them are unimportant to the average programmer, but a few of them are important, so I’ll cover them now.

Required Targets

If the requestor couldn’t know what data formats the owner supported, how did it know that it could get a list of those formats by requesting the value of the selection as TARGETS? Well, to make this all work, there have to be some rules, and one of those rules specifies a minimum set of targets that must be supported by the PRIMARY selection owner. The targets that must be supported are:

1. TARGETS. As mentioned above, it is vital for a pasting client to know the formats the data is available in. Thus a selection owner must respond to a request for TARGETS with a list of atoms identifying the formats it can provide.
2. TIMESTAMP. The owner of a selection must support conversion of the selection to a target of TIMESTAMP. The response to such a

request should be the value of the timestamp the owner used to acquire the selection.

3. MULTIPLE. I'm not going to go into any detail on this one except to say that if you implement cut and paste with the API provided by the Xt Intrinsics, this target is handled for you automatically.

Communications Magic

You may also be wondering how Client B's requests for the selection value got correctly routed to Client A. This detail is managed by the X server which maintains a table of selection atoms and their owners so that it can route selection requests to the proper client.

If a selection request is made for a selection that is not currently owned by any client, the server sends the requestor an event to indicate this.

Scenario 2: Losing My Selection

A selection owner can choose to relinquish ownership itself, or it can have the selection wrested from it when another client asserts ownership of that selection. Let's look at each of these cases in turn.

The first case is simple enough. It begins, like our last scenario, with the user making a gesture to select some data in client A. In response, client A highlights the data and asserts ownership of the PRIMARY selection.

Let's ignore for the moment whether or not the user actually ever does anything with this selected data. At some point in the future she makes another gesture to indicate that the data is no longer to be selected. Client A unhighlights the data and gives up selection ownership.

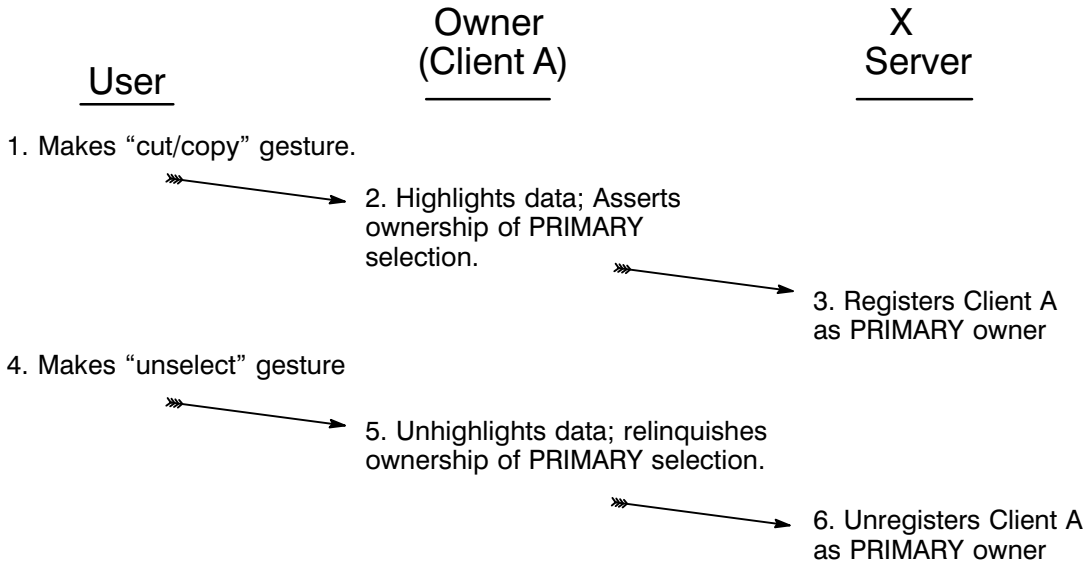


Figure 2: A client gives up selection ownership

A client also loses ownership if another client asserts ownership of that same selection. In this case, the user first selects some data in client A; she may or may not paste this data into another application. Eventually she selects some data in client B. At that point, the X server notifies client A that it no longer owns the selection and thus should unhighlight the old selected data.

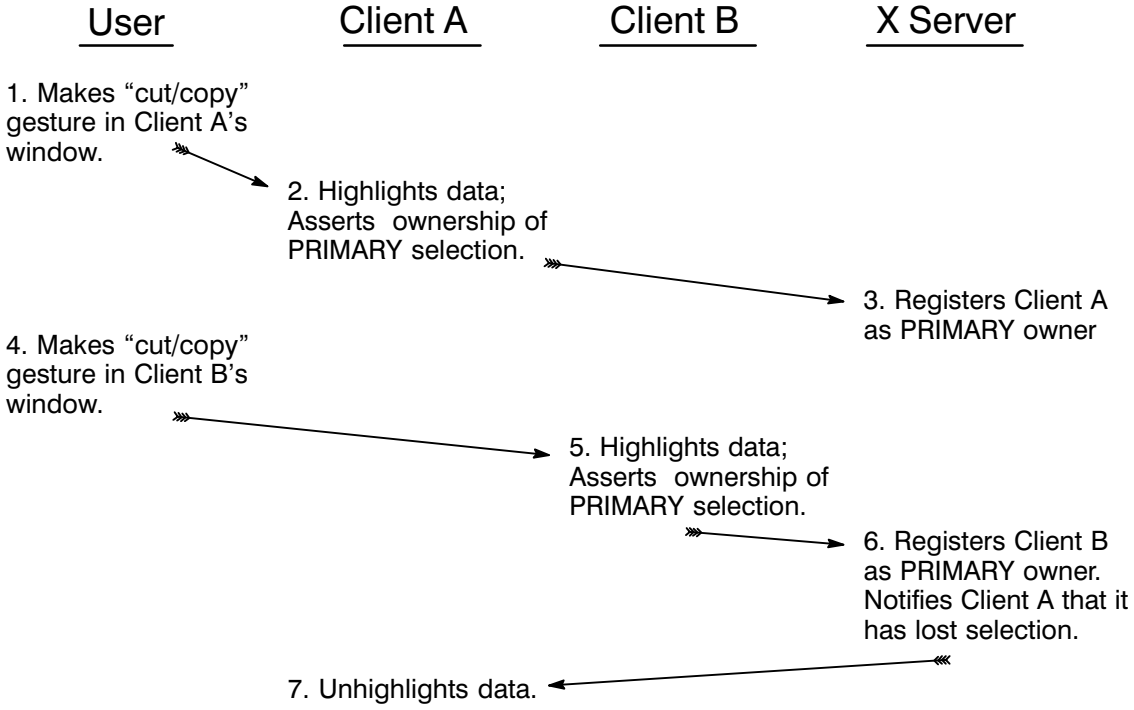


Figure 3: A client loses selection ownership involuntarily

PRIMARY vs. CLIPBOARD

As mentioned in the beginning of this section, there are two selections available for cut and paste in X: PRIMARY and CLIPBOARD. To this point, we've been discussing PRIMARY. Traditionally, the administrators and implementors of X have endeavored to provide mechanism, not policy. Since to only provide the PRIMARY selection would have, in essence, dictated a particular policy for how cut and paste work, the creators of the ICCCM also threw in CLIPBOARD. I guess they thought that giving people a choice of two policies was somehow better than dictating a single one. However, the competition between these two ways of doing things leads to some sticky interoperability issues.

Let's examine the two models in turn.

Global Primary Selection

This is the selection process described earlier in this section. The idea is that there is a single selected piece of data per X server. This data is visible and highlighted. If the user selects a piece of data in another application, the previously selected data is automatically unselected. Obviously, if the application containing the selected data terminates, the data is no longer available to other applications.

Primary selections are typically made ("cut") and inserted ("paste") by a mouse gesture.

Clipboard

This model presumes the user may want to have data selected in each of his applications, and that when he wants to make that data available to other applications, he will explicitly request a "cut" or "copy". Theoretically, such a request causes the data to be copied to a buffer independent of the application so that the data can remain available even after the application has terminated. Such a buffer might be maintained by a separate "clipboard" client that assumes responsibility for maintaining cut or copied data.

Typically, a user selects the data with a mouse gesture and then copies it to the clipboard either by pressing an accelerator key or choosing a menu item. Likewise, the data is pasted via accelerator key or menu item.

This is the model used on the Apple Macintosh and by Microsoft Windows, and is the model that purists are referring to when they use the term "cut and paste". (I don't know what term they expect you to use to refer to both models collectively.)

An Example

The Motif Text widget embodies both of these paradigms. When you select some text by dragging the mouse with Btn1 pressed, ownership of the PRIMARY selection is asserted and other applications can use the process outlined earlier to get the value of that selection. Similarly, a user can insert the value of the PRIMARY selection into the Text widget with Btn2.

Unbeknownst to most users, you can also copy selected data from the Text widget to the CLIPBOARD selection. Once you've used the mouse to select some text, you can use the Cut or Copy keys to put that data on the clipboard. Data from the CLIPBOARD selection can be inserted with the Paste key.

There are two key points that distinguish use of the CLIPBOARD selection from use of the PRIMARY selection.

1. It is a two-step process for the user to get data into the CLIPBOARD selection (select the data, then cut or copy it with another action).
2. The data is intended to be copied to a buffer that exists independently of the source application.

CLIPBOARD Scenario

Let's take a look at a successful transaction using the CLIPBOARD selection in the same way that we examined the process of using the PRIMARY selection. This scenario presumes that a separate clipboard client is running.

When a clipboard client starts up it asserts ownership of the CLIPBOARD selection. Subsequently, when another client asserts ownership (thus indicating it has data to be copied), the clipboard client is notified by the server that it has lost ownership of the selection. This instigates a request by the clipboard client for the value of the selection. Once it has obtained the selection data, the clipboard client reasserts ownership, so that any future requests for the data are directed to the clipboard client instead of the application the data was copied from.

The actual protocols for exchanging the data are the same as for PRIMARY. The difference is the insertion of an intermediary (the clipboard client) between the client where the data is cut and the client where the data is pasted.

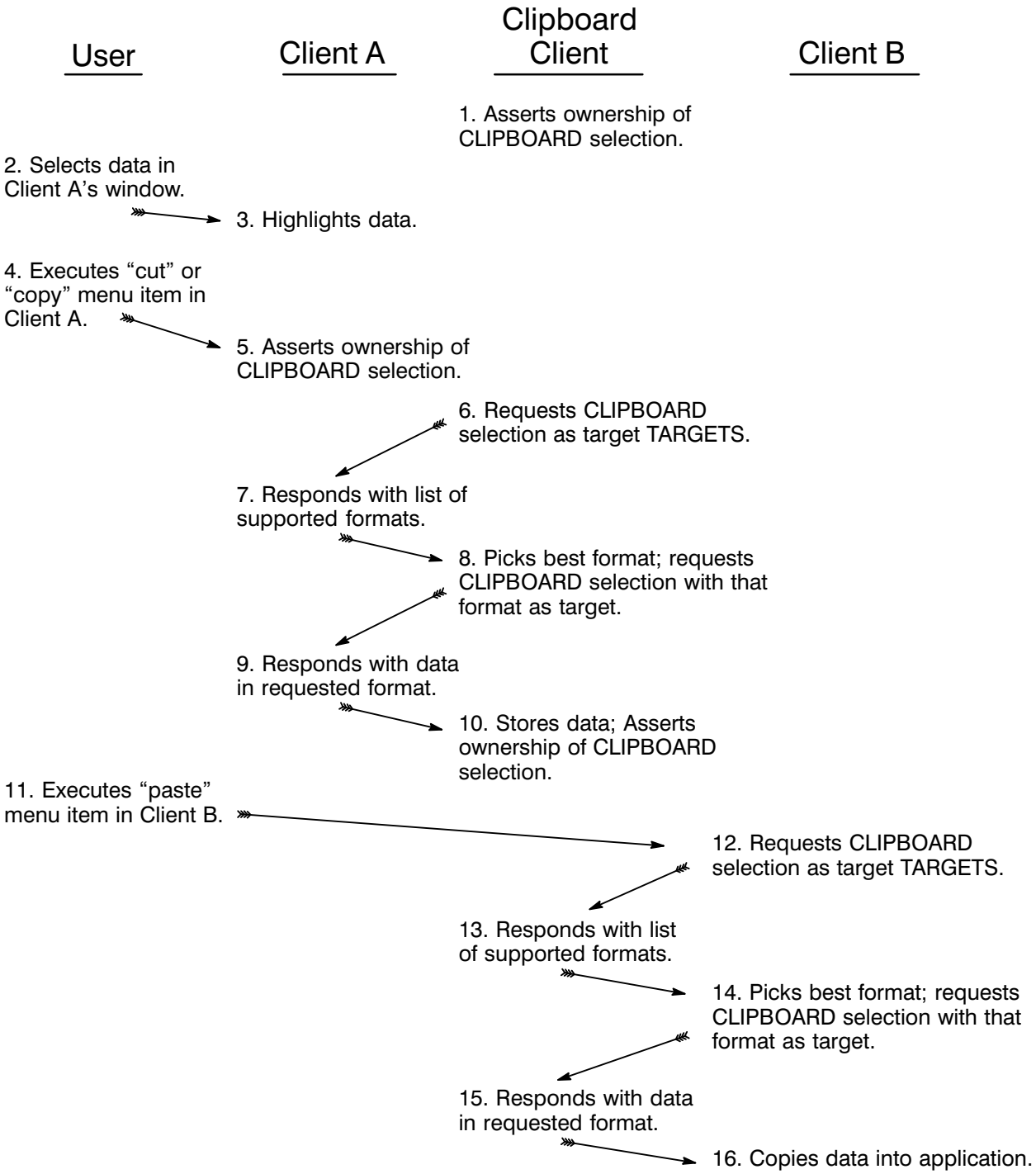


Figure 4: Exchanging data with the CLIPBOARD selection.

The Cold Hard Truth

Unfortunately, reality is not as simple as this nice neat diagram might lead you to believe. In practice, there are some problems with the CLIPBOARD selection model.

The ostensible advantage of the clipboard model is that it provides a global buffer (clipboard) in which applications can store the data the user has cut or copied. Once the data is stored there, the source application is out of the loop; it no longer has to worry about coughing up the data when the user pastes it into another application.

As seen in Figure 4, this global buffer is supposed to be provided by a separate clipboard client. But I am not aware of an implementation of such a client that fulfills all the necessary requirements. The one implementation I'm familiar with (`xclipboard`) only handles ASCII data. If an application tries to transfer any other type of data via the CLIPBOARD selection while `xclipboard` is running, that data will never get transferred and the user will not be able to paste it into another application.

Alternatively, the OSF/Motif library provides a clipboard API that implements an independent clipboard buffer without the need for a separate clipboard client. The problem with this approach is that only clients which use the Motif library will realize all the benefits of the clipboard approach (such as cut data persisting after the source client has terminated). While the Motif library is useful and popular, I think it is unrealistic to force developers to use it just to get clipboard-style cut and paste to work with other applications.

In addition to these practical problems, there is at least one serious liability inherent in use of the CLIPBOARD selection as defined by the ICCCM. When running a separate clipboard client, it chooses the data format the source client should use to transfer the data (Step 8 in Figure 4). But the clipboard client has no idea what data formats are useful to the program where the data ultimately gets pasted. Thus, the pasting client may get less useful information than it would have if it had picked the transfer format itself. Or, even worse, the pasting client may not be able to use the format chosen by the clipboard client, but could have used another format supplied by the source program.

For instance, suppose the user selects and copies a graphical object from Client A. Client A can supply representations of that object as a pixmap or as Encapsulated PostScript (EPS). The clipboard client chooses Encapsulated PostScript because that's a richer representation. However, when the user goes to paste the data into Client B, the operation fails because Client B does not support importation of EPS. Client B is capable of pasting pixmap data, so the transfer could have been made had the clipboard client not gotten in the way.

If a separate clipboard client is not running, you won't have this problem, but in that case the CLIPBOARD selection does not provide any more functionality than PRIMARY.

What's a Poor Developer To Do?

In the face of these obstacles, what should an application developer do to insure cut and paste interoperability with other applications? The difficult part is implementing flexible cut and paste in a way that does not confuse the end user. At a minimum, it seems the user needs to know that two different selections are available. I'm grateful that I'm not the one who needs to explain it to them, but let me propose a few potential approaches an application developer might take to bridge the schism between CLIPBOARD and PRIMARY.

PRIMARY Si, CLIPBOARD No

The "solution" requiring the least amount of effort is simply to forget about using the CLIPBOARD selection and only support data transfer through the PRIMARY selection. This is the approach used by the `xterm` terminal emulator program.

The positive aspects of this approach:

- Simplicity in programming. You only worry about owning or requesting one type of selection.
- A simple model for the user to understand. She doesn't have to worry about which gestures operate on which selections. PRIMARY is used consistently.

The negative aspects:

- Your application will not be able to paste from or cut to an application that only supports CLIPBOARD.
- If a robust, useful clipboard client is ever written, the user will not be able to use it in conjunction with your application.

The opposite approach (Supporting CLIPBOARD, but not PRIMARY) has similar positive aspects. However, since there are probably more applications that support only PRIMARY than there are applications that support only CLIPBOARD, I think this approach is somewhat less attractive in the near term.

Different Strokes for Different Selections

You could follow the example set by the Motif Text widget and support both PRIMARY and CLIPBOARD selections, with different gestures for manipulating them.

Advantages:

- Can interoperate with any application regardless of whether that application uses CLIPBOARD or PRIMARY.
- Follows a semi-sanctioned precedent by mimicking the Motif Text Widget.

Disadvantages:

- Slightly more programming required than just supporting a single selection.
- User must be aware of the two selections and the fact that different gestures are used to access them. They will have to know things like "when pasting data from xterm, use Btn2, but when pasting graphical data from FrameMaker, use the Paste menu item".
- Not all applications have a convenient way to provide gestures for accessing both CLIPBOARD and PRIMARY.

This seems to be a fairly reasonable approach. Since you are providing access to both selections, your application will be able to exchange data with any other application.

If you go this route, I suggest you use the same key bindings for the cut, copy, and paste gestures as the Motif Text widget in order to promote user interface consistency.

The Universal Approach

If you don't wish to complicate the user's life by providing two sets of gestures for cutting and pasting, but still want to be able to interoperate with any other application regardless of whether it uses CLIPBOARD or PRIMARY, you might want to consider the following strategy which I've dubbed "The Universal Approach".

Any time the user cuts or copies some data, you assert ownership of both the PRIMARY and CLIPBOARD selections. When the user requests a paste operation, if there is an owner for the PRIMARY selection, you use that; otherwise use the CLIPBOARD selection. Precedence is given to the PRIMARY selection because if a clipboard client is running, there will nearly always be a CLIPBOARD selection available. If you gave precedence to CLIPBOARD, your application would never be able to access data from the PRIMARY selection. There still is a potential problem, however. If a PRIMARY selection exists, but the data the user wants to paste is in CLIPBOARD, he will not get the expected results until he unselects the PRIMARY selection.

The advantages of this strategy:

- User does not have to distinguish between PRIMARY and CLIPBOARD when cutting or pasting to/from your application.
- Can interoperate with any application regardless of whether that application uses CLIPBOARD or PRIMARY.

Disadvantages:

- Programming is more complicated. For instance, you must be able to handle losing ownership of the selections individually.

- Can confuse the user in certain situations (like the one mentioned above where a PRIMARY selection exists, but the user really wants to paste the CLIPBOARD selection).

Conclusions

Which of these alternatives do you choose? Personally I think the second option (separate gestures for PRIMARY and CLIPBOARD) is the most attractive, with the slight drawback of requiring more intelligence on the part of the user. If your resources for implementing cut and paste are limited, then perhaps you should only support the PRIMARY selection. Finally, if you want your program to only have one set of cut and paste gestures, but still want to communicate with the widest range of applications, you should explore the last option described (the "universal" approach).

A Word About Cut Buffers

Selections did not exist in X10. Interclient data transfer was done via **cut buffers** (named data buffers accessible via the X server). Cut buffers and functions for accessing them were carried on into X11. Even though the use of cut buffers is denigrated by most X authorities, some programs continue to use them, primarily for compatibility with older applications that have not been updated to use selections.

If you need to interoperate with an older program that can only exchange data via cut buffers, then obviously you should add cut buffer support to your program (but you won't learn how to do that here). Otherwise, stick with selections.

3 Practice

Overview

Now that you've been saturated with the theory behind cut and paste in the X environment, let's turn our attention to the practical application of that theory. This section starts with an overview of the functions provided by the Xt Intrinsics that facilitate use of selections. This overview is followed by a detailed analysis of a sample program that uses these functions.

The Xt Selection API

There are a lot of good reasons to use a toolkit based on the Xt Intrinsics, not the least of which is the convenient high-level interface it provides for dealing with selections. The last section probably gave you an idea of how complicated the process is, and to be frank, programming it correctly in Xlib is not a task I would relish. Fortunately for those of us who can't cut the mustard, there is an easier way; you don't necessarily have to be a "hot-dog" programmer.

This summary of the API is not meant to be exhaustive. There is plenty of existing documentation for these functions; I only attempt to provide enough information about each call for you to understand its use in the sample program.

Owner Functions

Let's look at the various Xt selection functions, starting with the ones used by selection owners (clients that wish to "cut" some data and make it available to other clients).

```
Boolean XtOwnSelection (w, selection, time, c
                        convertCB, loseCB, doneCB);
Widget    w;
Atom      selection;
Time      time;
XtConvertSelectionProc  convertCB;
XtLoseSelectionProc     loseCB;
XtSelectionDoneProc     doneCB;
```

When a client wishes to obtain ownership of a selection, it calls `XtOwnSelection`, specifying a widget id, the selection it wishes to own, the timestamp from the user event that motivated the request, and callback routines to be called at various stages of the transfer process. If ownership of the selection is acquired, the function returns `True`, otherwise `False`. The first of the functions specified in the argument list, `convertCB`, is called to handle requests for the selection; `loseCB` is called when the client loses ownership of the selection; `doneCB` (when not `NULL`) is called to notify the client that a transfer has completed.

The `convertCB` function returns True or False based on whether it was able to convert the selection to the specified target and sets a pointer to where the data is stored. How this data storage is allocated depends on whether or not you specified a `doneCB` in the call to `XtOwnSelection`.

If you specified `doneCB` as `NULL`, then Xt assumes that your `requestCB` allocated space for the data with `XtMalloc` and will automatically free it with `XtFree` when the transfer completes. Otherwise, you are responsible for freeing the data (presumably in your `doneCB` routine). Most programs are probably best written with `doneCB` specified as NULL. Although there may be times when you can get away with just returning a pointer to the data "in place", rather than allocating a special area for the transfer, this will require more bookkeeping to avoid memory leaks. It also makes it trickier to handle multiple simultaneous selection requests.

Details on the arguments passed to each of these callback routines are provided in the analysis of the sample program.

```
XtDisownSelection (w, selection, time)
Widget   w;
Atom     selection;
Time     time;
```

When a client decides that it no longer wishes to own a selection, it calls `XtDisownSelection`, specifying the same `widget` and `selection` it specified in the original call to `XtOwnSelection`. For the `time` argument, you should pass the timestamp from the event that triggered the application to disown the selection. If the client did in fact own the selection, calling `XtDisownSelection` will cause the `loseCB` function registered by `XtOwnSelection` to be called.

Requestor Functions

There are two Xt functions for obtaining the value of a selection (and the second one is just a slight variant of the first).

```
XtGetSelectionValue (w, selection, target, valueCB,
                     client_data, time)
Widget     w;
Atom       selection;
Atom       target;
XtSelectionCallbackProc   valueCB;
XtPointer  client_data;
Time       time;
```

To obtain the value of a selection, a client calls `XtGetSelection-Value`, specifying a widget (any widget will do), the name of the selection, the data format (target) it wishes to have the selection converted to, a callback routine to call when the data is received and the timestamp from the event that motivated the request. You can also pass in a pointer that will be subsequently passed to the callback routine along with other arguments describing the selection data (again, these will be detailed in the analysis of the sample program).

```
XtGetSelectionValues (w, selection, targets, count, valueCB,
                      client_data, time)
Widget     w;
Atom       selection;
Atom       *targets;
int        count;
XtSelectionCallbackProc   valueCB;
XtPointer  client_data;
Time       time;
```

`XtGetSelectionValues` works the same as `XtGetSelection-Value`, except that you specify a list of targets to convert the selection to. The `valueCB` is called once for each target. Although it's not ap-

parent from the above declaration, `client_data` is a list of pointers, one for each target.

One benefit of using `XtGetSelectionValues` instead of calling `XtGetSelectionValue` repeatedly is that it guarantees that all the conversions will use the same selection value because the ownership of the selection cannot change in the middle of the list.

Help from Xmu

In addition to the functions provided by the Xt Intrinsics, the Xmu utility library also contains at least one useful function for dealing with selections.

```
extern Boolean
XmuConvertStandardSelection(w, time, selection,target, type,
                            value, length, format)
    Widget              w;
    Time                time;
    Atom                *selection;
    Atom                *target:
    Atom                *type_return:
    caddr_t             *value_return;
    unsigned long       *length_return;
    int                 *format_return;
```

`XmuConvertStandardSelection` can convert a selection to the following standard targets: CLASS, CLIENT_WINDOW, HOSTNAME, IP_ADDRESS, NAME, OWNER_OS, TARGETS, TIMESTAMP, and USER. Using this function is an easy way for your application to support a wider range of selection targets.

API for Incremental Transfers

The ICCCM defines a protocol to be used for transferring the value of a selection when there is a significant amount of data involved. This is known as an incremental transfer. The decision to transfer the selection incrementally is made by the selection owner. The Xt functions for requestors described above will automatically handle an incremental transfer.

For selection owners that wish to transfer selection data incrementally, there is the function `XtOwnSelectionIncremental`. This function is not covered by this document. See your Xt Intrinsics documentation for more information.

The Sample Program

I had hoped to produce a program to demonstrate use of the Xt API for cut and paste by modifying an existing program. However, I was unable to find a program that could be easily modified to suit my purposes, so I wrote one from scratch. A listing of this program, `display_text`, can be found in Appendix A along with instructions for obtaining an electronic version.

Let me make it clear from the start that the sole purpose of this program is to serve as an example for how to properly use the Xt selection functions. The other parts are not particularly stunning.

`display_text` does what it says it does: displays a string in a window. A user can select a range of the displayed text to become the PRIMARY selection by pressing Btn1 and dragging. Or they can append the PRIMARY selection to the end of the displayed text by pressing Btn2.

The sample program does not support the CLIPBOARD selection. Such support would be simple to add, but I didn't want to clutter the program for the purposes of this analysis. Thus, adding support for CLIPBOARD is left as an exercise for the reader.

In the ensuing discussion, numbers appearing in brackets, like [428], refer to program line numbers. I have refrained from reproducing portions of the code here; I expect you will have to flip back and forth between the text here and the program listing in Appendix A.

The structure of the program is straightforward. It creates a Motif DrawingArea widget as the only child of a MainWindow widget which is directly descendent from the toplevel shell [149–160]. The string is drawn in the DrawingArea in response to expose events [203]. Actions and translations to support selecting text and pasting the PRIMARY selection are defined [56–67] and added to the DrawingArea [162–165].

Cutting

The action routines for selecting a range of text to be the PRIMARY selection are `start_select` [378], `drag_select` [399], and `end_select` [435]. These handle the inverting of the text as it is selected. (I do not consider my code to be a good example of how to go about doing this, but it works.) `end_select`, which is called to complete the selection makes the call to `XtOwnSelection` that asserts ownership of the PRIMARY selection.

This call also registers the function `selection_requestCB` [494] as the callback function the Xt Intrinsics will call to process a request

for the selection value. As promised, let's take a closer look at this function and the arguments it is called with.

Three arguments are passed in: a widget id (the one specified in the call to `XtOwnSelection`), a pointer to an Atom identifying the selection whose value is being requested, and a pointer to an Atom identifying the target the value is to be converted to.

There are also four arguments that are pointers to items to be filled in by the callback: an Atom for the type of the returned data, the data itself, the length of the data, and the format. (The value returned for the format should be either 8, 16, or 32; this specifies the number of bits in a unit of the value and is used to determine what byte-swapping might be done when transferring the data over the connection between the client and the X server.)

The value returned by our callback is significant. If we successfully converted the selection to the requested target, we return `True`, otherwise, `False`.

The first thing `selection_requestCB` does is to call `XtGetSelectionRequest`. We use the request ID it returns to obtain a timestamp to pass to the Xmu convenience functions.

The rest of the function consists of four separate clauses for handling the various targets we support.

For TARGETS, we get the list of targets that Xmu handles for us, add the targets we handle, and then return the list. Note that we malloc space for the entire list and then copy in the Xmu targets. Since we specified `NULL` as the `doneCB` in the call to `XtOwnSelection`, the Intrinsics will free this data after the transfer is complete. The type of the data is ATOM and the format is 32 (the number of bits in an Atom).

The next clause handles the case where the requestor has asked to convert the selection to either TEXT or STRING. You might wonder what the difference between these is. According to the ICCCM, STRING gives you a string in ISO Latin 1 encoding, and TEXT is in the owner's encoding. Since our application's encoding is ISO Latin 1, we return the same value for both of these targets. The code here couldn't be much simpler: we just allocate some space, copy in the selected characters and return a pointer to the allocated space. The return length is set to the number of selected characters. For these two targets (TEXT and STRING), the atom returned for the type of data is the same as target itself, and since the return data is characters, we set the format to 8 (8 bits in a byte).

One optional target mentioned in the ICCCM is LENGTH. A conversion to this target should return the length of the selection in bytes. This raises the question "If I can return the selection in multiple for-

mats, which one are we talking about when we talk about the selection's length?" That dilemma doesn't occur in the example program, so I decided to support LENGTH to demonstrate handling an alternative target. The value returned is the number of selected characters; the type is INTEGER, and the format is the size of an `int` times 8.

If the requested target is not one of these, then it's either a target handled by Xmu, or one we don't handle at all. So we pass it on to `XmuConvertStandardSelection`, which will either do the conversion and return `True`, or return `False` if it couldn't do the conversion.

The other function involved in the process of owning the selection is `lost_ownershipCB` [583]. This gets called if our program loses ownership of the selection, either willingly by calling `XtDisownSelection`, or because another client has asserted ownership. This function redraws the range of selected text unhighlighted and resets the selection state variables to their default values.

Speaking of `XtDisownSelection`, it is worth noting the call to this function that appears in `end_select` [449]. If we already own the selection and the user clicks Btn1 in our window without dragging over any text, we intentionally disown the selection. Why own the selection if we don't have any data to hand out when someone asks for it? Other implementors, most notably the authors of the Motif 1.2 Text Widget, disagree and hold on to the selection even if there is no selected item.

Pasting

A request to paste the value of the PRIMARY selection is initiated by the function `paste_primary` [615]. It calls `XtGetSelectionOwner` with a request to convert the PRIMARY selection to target TARGETS. It sets up a callback (`targetCB`) to be called when the response to that request is received. We also specify that the `client_data` argument passed to that callback is the timestamp of the user event corresponding to the paste request.

`targetCB` [629] is the second stage of the pasting process. Ideally, it picks the best target from the list of targets supported by the owner and then asks to have the selection converted to that target. Let's go through it, step by step.

The first thing is to check if Xt returned a type argument indicating that the operation timed out. If so, we post an error dialog and return.

Next it checks if the type is `None`. Convention says that this means the conversion request failed. All this means is that the selection owner doesn't support the TARGETS target; this is not an unusual situation, so instead of failing completely we just decide to ask for a conversion to STRING. I suppose if I was just trying to write this program in the

quickest and dirtiest way I could have just done this in the first place and not bothered to find out what targets the owner supports. However, I'm trying to simulate what a "real" program will have to do, which presumably includes the ability to paste several formats (and consequently the need to choose the best available one).

On the other hand, if we do get a list of targets, and TEXT and STRING aren't in it, then we have to give up and post an error dialog to that effect.

Otherwise, once we've figured out which target to ask for, we make another call to `XtGetSelectionValue` with the desired target and the timestamp we got passed as `client_data`. The callback for this request is `valueCB` [665].

This function is straightforward. A few possible error conditions are checked at the beginning and if everything went as planned, the characters from the selection are appended to the end of the display string and then drawn. Finally, our callback is responsible for the space allocated for the selection data, so when we're done, we free it with `XtFree`.

Miscellaneous

There are a few other things in the program worth pointing out.

Atoms are used to name selections and targets. Some atoms are predefined by the X protocol (and have associated `#defines`) and some are not. Rather than worry about which atoms are predefined, I just declare a variable for every atom I'm going to use and then intern them all at the beginning of the program.

Note that every time a valid range of text is selected in `end_select`, we assert ownership of the selection, even though we may already have owned it. You might be tempted to optimize this call out, but it's very important to call `XtOwnSelection` every time the user selects something in order to keep the timestamp associated with the selection up to date.

There is at least one bug in the program. Although it will handle a request for converting the selection to the target MULTIPLE (through the magic of the Xt functions), that target is not included in the list of targets returned in response to a request for TARGETS. If this minor flaw bothers you, consider fixing it another exercise for the reader.

4 Targets

Rationale / Background

Any scheme for data exchange needs a method for identifying the meaning and format of the data to be exchanged. The selection protocol defined in the ICCCM uses *targets* for this purpose. A target is an atom that names the form the selection should be converted to before transfer.

A target is not strictly the same as a type. For instance, you might ask for the value of a selection to be converted to the target TIMESTAMP. TIMESTAMP is the target, but the type of the returned data is INTEGER. On the other hand, if you ask for a conversion to STRING, the type of the data will also be STRING. This gets a bit confusing because (to my mind) the concept of target is overloaded.

We can ease the confusion slightly by dividing targets into three categories:

1. Those that describe some secondary characteristic of the selected data and/or the owning client. This category includes TARGETS, TIMESTAMP, HOSTNAME, USER, CLASS, and NAME.
2. Those that identify a data format for transferring the actual selected data. These include TEXT, PIXMAP, and _ADOBE_EPS.
3. Those that specify an action (side-effect) to be performed on the selected data, such as DELETE and INSERT_SELECTION.

Standardization

It is imperative that there be agreement on a standard set of targets (as well as the meaning of those targets) to be used for selections. As previously mentioned, the ICCCM specifies three required targets for selection owners: TARGETS, MULTIPLE, and TIMESTAMP. The ICCCM also contains an “Initial Set of Target Atoms and their Meanings”. Unfortunately, this list is rather short and terse, and does not

seriously address what I consider the most urgent need for standardization: formats for non-textual data.

What mechanisms are there to facilitate this standardization?

X Registry

The X Consortium maintains a registry of X-related items, including selection targets. Any interested party can have an entry added to this list. To obtain more information about the X Registry, send an e-mail message with a subject of `send docs registry` to `xregistry@expo.lcs.mit.edu`. If you submit a selection target to the registry, I strongly suggest that your submission include the name of a document that rigorously defines the data format your target identifies. If you want to encourage people to support your target, you must provide an unambiguous reference for them to work from.

Existing Standard Targets

While the list is not extensive, there are a number of existing standard targets. Let's take a closer look at each of them, starting with those that specify some secondary aspect of the selection data or owner (Category 1 as defined above).

Category 1 (Secondary Aspects)

Name	Type	Originator
TARGETS	ATOM	ICCCM

We've already thoroughly discussed TARGETS. One thing worth noting though, is that `XmuConvertStandardSelection` supports conversion to TARGETS.

Name	Type	Originator
MULTIPLE	ATOM_PAIR	ICCCM

Make multiple conversion requests in one fell swoop. Selection owners using the Xt Intrinsics API automatically get conversion to this target.

Name	Type	Originator
TIMESTAMP	INTEGER	ICCCM

Return the timestamp used to obtain ownership of the selection. Selection owners using the Xt Intrinsics API automatically get conversion to this target.

Name	Type	Originator
LENGTH	INTEGER	ICCCM

Return the length of the selection in bytes. As discussed in the analysis of the sample program, the definition of this target is flawed in that you can't know which of the potentially numerous formats supported by the owner is being referred to.

Name	Type	Originator
HOSTNAME	TEXT	ICCCM

Hostname of the machine where the client program is running. `XmuConvertStandardSelection` supports conversion to HOSTNAME. Interestingly enough, the ICCCM lists this target as HOST_NAME, but since the Xmu implementation uses HOSTNAME, I have listed the name as such.

Name	Type	Originator
IP_ADDRESS	NET_ADDRESS	Xmu

Similarly, this one returns the IP address of the host where the client is running. I don't know where this target originated, it does not appear in either the ICCCM or the X Registry. But I included it since it's supported by `XmuConvertStandardSelection.`

Name	Type	Originator
USER	TEXT	ICCCM

Return the name of the user running the client. Supported by `XmuConvertStandardSelection`.

Name	Type	Originator
CLASS	TEXT	ICCCM

Return the WM_CLASS property of the client program. Supported by `XmuConvertStandardSelection`.

Name	Type	Originator
NAME	TEXT	ICCCM

Return the WM_NAME property of the client program. Supported by `XmuConvertStandardSelection`.

Name	Type	Originator
CLIENT_WINDOW	WINDOW	ICCCM

Return the window id of the client program's toplevel window. Supported by `XmuConvertStandardSelection`.

Category 2 (Data Formats)

These targets identify a specific data format in which the selection is to be transmitted.

Name	Type	Originator
STRING	STRING	ICCCM

Textual data encoded as ISO Latin 1.

Name	Type	Originator
COMPOUND_TEXT	COMPOUND_TEXT	MIT X Consortium

Compound text encoded as described in the proposal for compound text representation by Robert Scheifler at MIT.

Name	Type	Originator
TEXT	*text*	ICCCM

Textual data encoded at the convenience of the selection owner. Note that the atom returned for the type should identify the encoding used. (Available encodings and their names are listed in the X Registry.)

Name	Type	Originator
PIXMAP	DRAWABLE	ICCCM

A pixmap id. Presumably graphical selection data has been rendered to the pixmap. Use of this selection target is problematic because if the client that created the pixmap terminates, the id will no longer be valid. To be safe, upon receiving the pixmap id, the requestor should immediately create its own pixmap and copy the contents from the original pixmap to this new pixmap. This is not entirely foolproof, but is far superior to just blindly using the original pixmap id.

Name	Type	Originator
BITMAP	BITMAP	ICCCM

Same as pixmap, but a single-plane.

Name	Type	Originator
_ADOBE_EPS	_ADOBE_EPS	Adobe Systems Inc.
_ADOBE_EPSI	_ADOBE_EPSI	Adobe Systems Inc.

Both of these targets identify data formatted in Encapsulated PostScript. To find out how to obtain official guidelines for creating Encapsulated PostScript send the one-word message "help" to `ps-file-server@adobe.com`. Or you can ftp to `ftp.adobe.com`. The difference between _ADOBE_EPS and _ADOBE_EPSI is that an EPSI file includes a bitmap preview image of the PostScript-encoded data, so that applications unable to directly render PostScript can at least display a simple representation of the data.

Name	Type	Originator
_WRI_XWD_RLED	[?]	Wolfram Research Inc.

Run length encoded xwd format. Documentation for this format and a sample program which uses it (including library code for generating the data from an XImage) is available via anonymous ftp from `ftp.x.org` in `/contrib/xwdclip.tar.Z`.

Name	Type	Originator
_SONY_XWD	[?]	Sony

Straight xwd format. This and some other target atoms were registered by Sony for use with their proprietary widget set, but are no longer in general use.

Category 3 (Side-effects)

The following targets specify actions to be performed by the owning client on the selected data. While these targets are "standard" in that they appear in the ICCCM, I am not aware of any programs that use them.

Also, as a programmer, I'm a bit wary of allowing other programs to perform possibly irreversible side-effects on the data in my program. If you're the same way, you might want to think twice before implementing these targets.

Name	Type	Originator
DELETE	NULL	ICCCM

Delete the selection and return a zero-length property of type NULL.

Name	Type	Originator
INSERT_SELECTION	NULL	ICCCM

Obtain the value of the selection named in the request and insert it at the location of the current owned selection.

Name	Type	Originator
INSERT_PROPERTY	NULL	ICCCM

Obtain the value of the property named in the request and insert it at the location of the current owned selection.

The Future

5 The selectinfo Program

Background / Rationale

Interoperability in the X environment requires cooperation between the clients sharing that environment. Lack of information can be a barrier to this cooperation. Imagine that you want your program to be able to exchange data with Lotus 1-2-3 via selections. How do you know what targets you will have to support to make this happen? That kind of information never appears in the documentation.

In the course of writing this guide, I frequently needed answers to questions like this. To simplify the process of researching exactly how various X applications support cut and paste, I wrote an application called `selectinfo` to display information about the current selection. I find it indispensable for diagnosing cut and paste idiosyncrasies, and thus worth sharing.

Program Overview

`selectinfo` is a Motif-based program that unobtrusively obtains important information about a selection and its owner. If the program is invoked with a name containing the string "clip", then it displays information about the CLIPBOARD selection, otherwise it displays information about the PRIMARY selection. The makefile provided creates two links to the same executable, one named `primary_info` and the other named `clipboard_info`.

Source for `selectinfo` can be obtained via anonymous ftp from `ftp.org.x` in the file `select_progs.tar.Z`.

Using selectinfo

This is a very simple program to use.

1. Invoke the program from the command line (either `primary_info` or `clipboard_info`). The window appears with blank data fields.

2. Perform the necessary steps to make a selection in the program of interest.
3. Click on the “Update” button at the bottom of the `primary_info` window. The data fields get filled in with the relevant values.

`selectinfo` displays the id of the window that owns the selection and the name of the application (determined by traversing up the window hierarchy until it finds a window with the WM_CLASS property set). If the owning client supports the TIMESTAMP target, then the timestamp used to acquire the selection is displayed; if it supports the target TARGETS, a list of available targets is displayed. If either TIMESTAMP or TARGETS is not supported by the owning client, `selectinfo` pops up an error dialog to that effect.

Probably the most useful information is the list of targets, since it tells you what data formats the application can export via selections. If you're investigating an application that allows the user to select different types of objects (for instance, a range of text versus a portion of a drawing), you should perform steps 2 and 3 above to see the selection info for each possible type.

Note that `selectinfo` is only able to provide information about the capabilities of a client as selection owner. While it is likely that a client will import the same formats that it exports, it is not certain. About the only method I can suggest for finding out this information is simple trial and error. Play around with changing the targets your program provides until you can tell which targets are requested by the application program under investigation.

If, through the use of `selectinfo`, you discover that a program is not adhering to the ICCCM (perhaps by not supporting the required targets), I urge you to contact the vendor or author of the program and let them know. Most X programmers want to follow the rules in the ICCCM, but sometimes things slip through the cracks.

A The Sample Program

For the purposes of following along with the analysis in Chapter 3, here's a listing of the sample program that implements cut and paste using the PRIMARY selection.

An electronic version of this source can be obtained via anonymous ftp from `ftp.org.x` in `select_progs.tar.Z`.

```
/************************************************************

     display_text.c  --  Simple program to demonstrate how to
        use Xt's API for selections.

Copyright 1993, Hewlett-Packard Company.

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and to
use, copy, and distribute its documentation for any purpose and without
fee is hereby granted, provided that the above copyright notice appear
in all copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Hewlett-Packard not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.

HEWLETT-PACKARD DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL HEWLETT-PACKARD BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.

************************************************************/
#include <stdio.h>

#include <X11/Xatom.h>
#include <X11/Intrinsic.h>
#include <X11/Xmu/StdSel.h>

#include <Xm/Xm.h>
#include <Xm/Protocols.h>
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>
#include <Xm/MessageB.h>

#define MAX_ARGS        25
#define MAX_TEXT_LEN    500
```

```
41
42    #define MIN(x, y)       ((x > y) ? y : x)
43    #define MAX(x, y)       ((x > y) ? x : y)
44
45    void            quitCB(), exposeCB();
46    Boolean         selection_requestCB();
47    void            lost_ownershipCB();
48    void            draw_char(), draw_range(), create_gcs();
49    void            start_select(), drag_select(), end_select();
50    void            paste_primary();
51    void            targetCB(), valueCB();
52    Atom            pick_best_target();
53    void            error_dialog();
54
55
56    static XtActionsRec actions[] = {
57            {"StartSelect", start_select},
58            {"DragSelect", drag_select},
59            {"EndSelect", end_select},
60            {"PastePrimary", paste_primary},
61    };
62
63    static char newTranslations[] =
64            "<Btn1Down>:    StartSelect()\n\
65             <Btn1Motion>:  DragSelect()\n\
66             <Btn1Up>:      EndSelect()\n\
67             <Btn2Down>:    PastePrimary()";
68
69    char            *progname;
70    char            the_text[MAX_TEXT_LEN+1];
71    Dimension       win_width, win_height;
72    int             char_width, char_height;
73    Boolean         own_primary = False;
74    int             first_select_char = -1,
75                    last_select_char = -1;
76
77    Display         *dpy;
78    int             screen;
79    XtAppContext    app_context;
80    Widget          toplevel;
81    Atom            xa_WM_DELETE_WINDOW;
82    Atom            xa_PRIMARY;
83    Atom            xa_TEXT;
84    Atom            xa_STRING;
85    Atom            xa_TARGETS;
86    Atom            xa_ATOM;
87    Atom            xa_LENGTH;
88    Atom            xa_INTEGER;
89    GC              normal_gc = NULL, inverse_gc;
90    char            *fontname = "9x15";    /* Should be a resource ??? */
91    XFontStruct     *fontinfo;
92
93
94    void main (argc,argv)
95    unsigned int    argc;
96    char            **argv;
97    {
98        Widget      main_win;
99        Widget      canvas;
100       Arg         arg[MAX_ARGS];
101       int         n;
102       XtTranslations      my_translations;
103
104       progname = argv[0];
105
106       /*
107          Initialize the toolkit, open the display, and create a
108          toplevel window.
109       */
110       XtToolkitInitialize();
111       app_context = XtCreateApplicationContext();
```

```
112         dpy = XtOpenDisplay (app_context, NULL, argv[0], "DisplayText", NULL, 0,
113                                  &argc, argv);
114         if (!dpy) {
115             XtWarning ("display_text: Can't open display, exiting...");
116             exit (0);
117         }
118         screen = DefaultScreen(dpy);
119
120         if (argc > 1) {
121             strncpy(the_text, argv[1], MAX_TEXT_LEN);
122         } else {
123             strcpy(the_text, "This is a default string because you didn't specify one
on the command line");
124         }
125
126         n = 0;
127         XtSetArg(arg[n], XmNdeleteResponse, XmDO_NOTHING); n++;
128         XtSetArg(arg[n], XmNgeometry, "=300x100"); n++;
129         toplevel = XtAppCreateShell (argv[0], "SelectOMatic",
130                                  applicationShellWidgetClass, dpy, arg, n);
131
132         /*
133            Intern some atoms we'll be needing
134         */
135         xa_WM_DELETE_WINDOW = XInternAtom(dpy, "WM_DELETE_WINDOW", False);
136         xa_PRIMARY = XInternAtom(dpy, "PRIMARY", False);
137         xa_STRING = XInternAtom(dpy, "STRING", False);
138         xa_TEXT = XInternAtom(dpy, "TEXT", False);
139         xa_TARGETS = XInternAtom(dpy, "TARGETS", False);
140         xa_LENGTH = XInternAtom(dpy, "LENGTH", False);
141         xa_ATOM = XInternAtom(dpy, "ATOM", False);
142         xa_INTEGER = XInternAtom(dpy, "INTEGER", False);
143
144         /*
145            Set it up so we find out when the window's closed
146         */
147         XmAddWMProtocolCallback(toplevel, xa_WM_DELETE_WINDOW, quitCB, NULL);
148
149         n = 0;
150         main_win = XmCreateMainWindow(toplevel, "main", arg, n);
151         XtManageChild(main_win);
152
153         /*
154             Create the Drawing Area for displaying the string,
155             and add our selection-related translations.
156         */
157         n = 0;
158         canvas = XmCreateDrawingArea(main_win, "canvas", arg, n);
159         XtManageChild(canvas);
160         XtAddCallback(canvas, XmNexposeCallback, exposeCB, NULL);
161
162         XtAppAddActions(app_context, actions, XtNumber(actions));
163         my_translations = XtParseTranslationTable(newTranslations);
164         XtSetArg(arg[0], XmNtranslations, my_translations);
165         XtSetValues(canvas, arg, 1);
166
167         XmMainWindowSetAreas(main_win, NULL, NULL, NULL, NULL, canvas);
168
169         XtRealizeWidget (toplevel);
170
171     #ifdef DEBUG
172         XtAppSetSelectionTimeout(app_context, 200000);
173     #endif
174
175         XtAppMainLoop (app_context);
176     }
177
178
179     /*
180             quitCB                  - callback for quit button
181     */
```

```
182     void quitCB (w, client_data, call_data)
183     Widget          w;              /*  widget id           */
184     caddr_t         client_data;    /*  data from application   */
185     caddr_t         call_data;      /*  data from widget class  */
186     {
187
188         /*
189             Terminate the application.
190          */
191         exit (0);
192     }
193
194
195
196     /*************************************************************
197
198                 Stuff for Displaying the String
199
200     *************************************************************/
201
202
203     void exposeCB (canvas, client_data, call_data)
204     Widget canvas;
205     XtPointer client_data;
206     XtPointer call_data;
207     {
208         Arg      arg[4];
209         Window   w = XtWindow(canvas);
210
211         /*
212             First time through, set some things up
213         */
214         if (!normal_gc) {
215             Pixel       fg, bg;
216
217             setup_gcs(XtWindow(canvas));   /* Assuming it works ??? */
218             XtSetArg(arg[0], XmNbackground, &bg);
219             XtSetArg(arg[1], XmNforeground, &fg);
220             XtGetValues(canvas, arg, 2);
221             XSetBackground(dpy, normal_gc, bg);
222             XSetForeground(dpy, normal_gc, fg);
223             XSetBackground(dpy, inverse_gc, fg);
224             XSetForeground(dpy, inverse_gc, bg);
225             XtAddCallback(canvas, XmNresizeCallback, exposeCB, NULL);
226         }
227
228         /*
229             Update our globals with the new width and height.
230         */
231         XtSetArg(arg[0], XmNwidth, &win_width);
232         XtSetArg(arg[1], XmNheight, &win_height);
233         XtGetValues(canvas, arg, 2);
234
235         /*
236             Refresh the string. This is *not* the most efficient way
237             to do this, but it was easy to code.
238         */
239         XClearWindow(dpy, w);
240
241
242         if (first_select_char != last_select_char) {
243             draw_range(w, 0, first_select_char, normal_gc);
244             draw_range(w, first_select_char, last_select_char, inverse_gc);
245             draw_range(w, last_select_char, strlen(the_text), normal_gc);
246         } else {
247             draw_range(w, 0, strlen(the_text), normal_gc);
248         }
249     }
250
251
252     /*
```

```
253         Draw the nth character of the string in Window w with GC gc.
254     */
255
256     void draw_char (w, n, gc)
257     Window    w;
258     int       n;
259     GC        gc;
260     {
261         int   numcols;
262         int   x, y;
263
264         if ((n >= 0) && (n < strlen(the_text))) {
265             numcols = win_width / char_width;
266             /*
267             x = ((n % numcols) * char_width) + fontinfo->max_bounds.lbearing;
268             y = ((n / numcols) * char_height) + fontinfo->max_bounds.ascent;
269             */
270             x = ((n % numcols) * char_width);
271             y = ((n / numcols) * char_height) + char_height;
272             XDrawImageString(dpy, w, gc, x, y, the_text + n, 1);
273         }
274     }
275
276
277     /*
278         Draw the characters char1 through char2 in Window w with GC gc.
279         This function will handle the args in any order, so the drawing
280         always includes the character with the lower index, but excludes
281         the character with the higher index.
282     */
283
284     void draw_range (w, char1, char2, gc)
285     Window   w;
286     int      char1, char2;
287     GC       gc;
288     {
289         int  tmp, i;
290
291         if (char1 > char2) {
292             tmp = char1;
293             char1 = char2;
294             char2 = tmp;
295         }
296
297         for (i = char1; i < char2; i++)
298             draw_char(w, i, gc);
299     }
300
301
302
303     /*
304         Open the font we'll be using and create a couple gc's.
305     */
306
307     setup_gcs (w)
308     Window   w;
309     {
310         fontinfo = XLoadQueryFont(dpy, fontname);
311
312         if (!fontinfo) {
313            fprintf(stderr, "Unable to open font %s\n", fontname);
314            exit (1); /* ??? */
315         }
316         char_height = fontinfo->max_bounds.ascent + fontinfo->max_bounds.descent;
317         char_width = fontinfo->max_bounds.width;
318
319         normal_gc = XCreateGC(dpy, w, 0l, NULL);
320         XSetFont(dpy, normal_gc, fontinfo->fid);
321         inverse_gc = XCreateGC(dpy, w, 0l, NULL);
322         XSetFont(dpy, inverse_gc, fontinfo->fid);
323     }
```

```
324
325
326
327     /************************************************************
328
329        User Interface Stuff for Selecting Text (highlighting, etc.)
330
331     ************************************************************/
332
333
334     /*
335         Translate a pixel coordinate into the index of the character
336         in the string that appears at that pixel coordinate.
337     */
338
339     int pixel_coords_to_char (x, y)
340     int   x, y;
341     {
342         int  char_x, char_y;
343         int  num_cols, num_rows;
344         int  length;
345         int  n;
346
347         /*
348             Do some range filtering
349         */
350         length = strlen(the_text);
351         num_cols = win_width / char_width;
352         num_rows = (length / num_cols) + 1;
353         char_x = x / char_width;
354         char_x = MAX(char_x, 0);
355         char_x = MIN(char_x, num_cols);
356         char_y = y / char_height;
357         char_y = MAX(char_y, 0);
358         char_y = MIN(char_y, num_rows);
359
360         n = (char_y * num_cols) + char_x;
361
362         if (n > length)
363             n = length;
364
365         return (n);
366     }
367
368
369     static int   done_first = False;
370     static int   last_drawn = -1;
371
372
373     /*
374         The user has pressed Btn1. Remember the character the button
375         was depressed over.
376     */
377
378     void start_select (w, event)
379     Widget   w;
380     XButtonEvent  *event;
381     {
382
383         /*
384            Erase the old selection highlight
385         */
386         draw_range(XtWindow(w), first_select_char, last_select_char, normal_gc);
387
388         done_first = False;
389         first_select_char = pixel_coords_to_char (event->x, event->y);
390         last_drawn = -1;
391     }
392
393
394     /*
```

```
395         Highlight the text the user is selecting appropriately.
396         Don't worry, even I'm not sure why it works.
397     */
398
399     void drag_select (w, event)
400     Widget    w;
401     XButtonEvent   *event;
402     {
403         int  next_select_char;
404
405         next_select_char = pixel_coords_to_char(event->x, event->y);
406
407         if (next_select_char == last_drawn)
408             return;
409
410         if (done_first) {
411             draw_range(XtWindow(w), last_drawn, first_select_char, normal_gc);
412             draw_range(XtWindow(w), first_select_char, next_select_char,
413                        inverse_gc);
414         } else {
415             draw_range(XtWindow(w), first_select_char, next_select_char,
416                        inverse_gc);
417             done_first = True;
418         }
419
420         last_drawn = next_select_char;
421     }
422
423
424     /*
425         Btn1 has been released. Remember the character the pointer was
426         over when the button was released.
427
428         Set first_select_char and last_select_char to point to the range
429         of selected text. The text is inclusive of first_select_char and
430         exclusive of last_select_char.
431
432         Assert ownership of the primary selection.
433     */
434
435     void end_select (w, event)
436     Widget    w;
437     XButtonEvent   *event;
438     {
439         int tmp;
440
441         last_select_char = pixel_coords_to_char(event->x, event->y);
442
443         /*
444            We didn't get any text selected.
445         */
446         if (last_select_char == first_select_char) {
447             /* Disown PRIMARY selection if we own it */
448             if (own_primary)
449                 XtDisownSelection(w, xa_PRIMARY, event->time);
450             first_select_char = last_select_char = -1;
451             return;
452         }
453
454         /* Get them in order */
455         if (last_select_char < first_select_char) {
456             tmp = last_select_char;
457             last_select_char = first_select_char;
458             first_select_char = tmp;
459         }
460
461         first_select_char = MAX(0,first_select_char);
462         last_select_char = MIN(strlen(the_text), last_select_char);
463
464         /*
465            Okay, now we've got something selected. Let's tell the world.
```

```
466         */
467         own_primary = XtOwnSelection(w, xa_PRIMARY, event->time,
468                                      selection_requestCB,
469                                      lost_ownershipCB,
470                                      NULL);
471         /*
472            Have to make sure we actually obtained ownership.
473            If we didn't, take steps to clear selection data.
474         */
475         if (!own_primary)
476             lost_ownershipCB(w, xa_PRIMARY);
477
478     }
479
480
481
482     /*************************************************************
483
484      Stuff for Managing Selection Ownership and Requests. (Cutting)
485
486     *************************************************************/
487
488
489     /*
490         Someone has asked us for the value of a selection.
491         Respond appropriately, based on the target they specified.
492     */
493
494     Boolean selection_requestCB (w, selection, target, type, value, length, format)
495     Widget  w;              /* In */
496     Atom    *selection;     /* In */
497     Atom    *target;        /* In */
498     Atom    *type;          /* Out */
499     XtPointer *value;       /* Out */
500     unsigned long *length;  /* Out */
501     int     *format;        /* Out */
502     {
503          int   i;
504          XSelectionRequestEvent   *req;
505          Atom       *tmp_target_list;
506          int        *tmp_length;
507
508     #ifdef DEBUG
509          printf("In requestCB, target is %s\n", XGetAtomName(dpy, *target));
510     #endif
511
512          req = XtGetSelectionRequest(w, *selection, NULL);
513
514         /*
515            First, handle requests for TARGETS
516         */
517         if (*target == xa_TARGETS) {
518             Atom   *xmu_targets;
519             unsigned long  xmu_length;
520
521             /*
522                Get Xmu to list targets it handles
523             */
524             XmuConvertStandardSelection(w, req->time, selection, target, type,
525                                         &xmu_targets, &xmu_length, format);
526             /*
527                 Put our targets in the list.
528             */
529             *length = xmu_length + 3; /* 3 is number of targets we support */
530             tmp_target_list = (Atom *)XtMalloc(sizeof(Atom) * (*length));
531             tmp_target_list[0] = xa_STRING;
532             tmp_target_list[1] = xa_TEXT;
533             tmp_target_list[2] = xa_LENGTH;
534             /*
535                 Copy the Xmu targets into our list
536             */
```

```
537             for (i = 3; i < *length; i++)
538                 tmp_target_list[i] = xmu_targets[i-3];
539             XtFree((char *)xmu_targets);
540             *value = (XtPointer)tmp_target_list;
541             *type = xa_ATOM;
542             *format = 32;
543             return True;
544         }
545 
546         /*
547             Handle a request for STRING or TEXT
548             Malloc space and copy the string in.
549             Just pass a pointer to the appropriate part of the string.
550         */
551         if ((*target == xa_TEXT) || (*target == xa_STRING)) {
552             *length = last_select_char - first_select_char;
553             *value = XtMalloc(*length);
554             for (i = 0; i < *length; i++)
555                 (*value)[i] = the_text[first_select_char + i];
556             *type = xa_STRING;
557             *format = 8;
558             return True;
559         }
560 
561         /*
562             Handle a request for LENGTH
563         */
564         if ((*target == xa_LENGTH)) {
565             tmp_length = (int *)XtMalloc(sizeof(int));
566             *tmp_length = last_select_char - first_select_char;
567             *value = (XtPointer)tmp_length;
568             *length = 1;
569             *type = xa_INTEGER;
570             *format = sizeof(int) * 8;
571             return True;
572         }
573 
574         /*
575             We couldn't do it, maybe Xmu can
576         */
577         return (XmuConvertStandardSelection(w, req->time, selection, target,
578                                             type, value, length, format));
579 
580     }
581 
582 
583     void lost_ownershipCB (w, selection)
584     Widget   w;
585     Atom     *selection;
586     {
587 
588         /*
589             If we lose ownership of primary, clean up our highlight and
590             reset our state.
591         */
592         if (*selection == xa_PRIMARY) {
593             draw_range(XtWindow(w), first_select_char, last_select_char, normal_gc);
594             done_first = False;
595             first_select_char = last_select_char = -1;
596             last_drawn = -1;
597             own_primary = False;
598         }
599     }
600 
601 
602 
603     /*********************************************************************
604 
605      Stuff for Obtaining Selection Values Upon User Request. (Pasting)
606 
607     *********************************************************************/
```

```
608
609
610     /*
611         User asked to get PRIMARY selection.
612         First find out what formats are available.
613     */
614
615     void paste_primary (w, event)
616     Widget   w;
617     XButtonEvent    *event;
618     {
619         XtGetSelectionValue (w, xa_PRIMARY, xa_TARGETS, targetCB,
620                              (XtPointer)event->time, event->time);
621     }
622
623
624     /*
625         Pick the best format from the list available and ask
626         for a conversion to that target.
627     */
628
629     void targetCB (w, client_data, selection, type, value, length, format)
630     Widget          w;
631     XtPointer       client_data;
632     Atom            *selection;
633     Atom            *type;
634     XtPointer       value;
635     unsigned long   *length;
636     int             *format;
637     {
638         Atom    best_target;
639         Time    paste_time = (Time)client_data;
640
641         if (*type == XT_CONVERT_FAIL) {
642               error_dialog("Paste operation timed out, try again.");
643               return;
644         }
645
646         if ((*type == None) || (*length == 0)) {
647             /* Could be that owner doesn't support TARGETS, just try STRING */
648             best_target = xa_STRING;
649         } else {
650             best_target = pick_best_target((Atom *)value, *length);
651         }
652
653         if (best_target == NULL)
654             error_dialog("No appropriate data to paste");
655         else
656             XtGetSelectionValue (w, *selection, best_target, valueCB,
657                                  NULL, paste_time);
658     }
659
660
661     /*
662         We got the data, let's tack it on to our string.
663     */
664
665     void valueCB (w, client_data, selection, type, value, length, format)
666     Widget          w;
667     XtPointer       client_data;
668     Atom            *selection;
669     Atom            *type;
670     XtPointer       value;
671     unsigned long   *length;
672     int             *format;
673     {
674         int   current_length;
675         int   chars_to_copy, i;
676         char  *tmp_ptr;
677
678         /*
```

```
679             Check various possibilities for error.
680         */
681
682         if (*type == XT_CONVERT_FAIL) {
683             error_dialog("Paste operation timed out, try again.");
684             return;
685         }
686         if (*type == None) {
687             error_dialog("No selection, or Paste operation failed.");
688             return;
689         }
690         if ((*type != xa_STRING) && (*type == xa_TEXT)) {
691             XtFree(value);
692             error_dialog("Attempt to paste unsupported data");
693             return;
694         }
695
696         current_length = strlen(the_text);
697         chars_to_copy = MIN(*length, MAX_TEXT_LEN - current_length);
698         tmp_ptr = the_text + current_length;
699         for (i = 0; i < chars_to_copy; i++) {
700             *tmp_ptr++ = *(value + i);
701         }
702         *tmp_ptr = '\0';
703
704         /* Draw the new part of our string */
705         draw_range(XtWindow(w), current_length, current_length + chars_to_copy + 1,
706                    normal_gc);
707
708         /* Let the user know if we dropped any characters */
709         if ((*length + current_length) > MAX_TEXT_LEN) {
710             error_dialog("Complete paste would've exceeded maximum buffer size.\
711      Partial paste performed.");
712         }
713
714         XtFree(value);
715     }
716
717
718     /*************************************************************
719
720                             Utilities
721
722     *************************************************************/
723
724     static Atom  *preferred_targets[] = {
725                                         &xa_STRING,
726                                         &xa_TEXT
727                                         };
728
729     /*
730         Go through our list of preferred targets and see if
731         they are supported by the owning client. Return the
732         first match.
733
734         If none match, return NULL.
735     */
736
737     Atom pick_best_target (target_list, n)
738     Atom  *target_list;
739     unsigned long   n;
740     {
741         int i, j;
742
743         for (i = 0; i < XtNumber(preferred_targets); i++)
744             for (j = 0; j < n; j++)
745                 if (target_list[j] == *preferred_targets[i])
746                     return (target_list[j]);
747
748         return NULL;
749     }
```

```
750
751
752     /*
753         Put up a dialog with the given message.
754     */
755
756     void error_dialog (str)
757     char  *str;
758     {
759         Widget    dialog;
760         Arg       arg[5];
761         XmString  tmpxmstr;
762         int       n;
763
764         n = 0;
765         tmpxmstr = XmStringCreateSimple(str);
766         XtSetArg(arg[n], XmNmessageString, tmpxmstr); n++;
767         XtSetArg(arg[n], XmNdeleteResponse, XmDESTROY); n++;
768         dialog = XmCreateMessageDialog(toplevel, "error_dialog", arg, n);
769         XmStringFree(tmpxmstr);
770
771         XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_CANCEL_BUTTON));
772         XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_HELP_BUTTON));
773         XtManageChild(dialog);
774
775     }
776
```

B Applications

In the process of writing this document I spent a fair amount of time examining how various X11 applications support cut and paste. Judging from the results, writing this document was effort well spent. Very few of the applications followed the rules to the letter.

However, I think knowing how other applications behave is useful to programmers who want to understand the requirements for making their programs interoperate with existing ones. When known, I have specified the application version number.

If you have access to newer versions or applications not listed here, please e-mail me (`chan@fc.hp.com`) updated information to be included in the next release of this document.

Interleaf 5.3.0

In Interleaf 5, selections are manipulated via the **X Paste** and **X Copy** menu items on the **Misc** submenu. The **X Copy** item is available when a range of text has been selected, otherwise you'll see **X Paste**.

When you select a range of text and execute **X Copy**, Interleaf asserts ownership of both the PRIMARY and CLIPBOARD selections, as well as inserting the selected characters into the X Cut Buffer 0. STRING is the only target that Interleaf will convert the selections to. It supports neither TIMESTAMP nor TARGETS.

When you execute X Paste, Interleaf will first ask for the CLIPBOARD selection as STRING. If that fails, it tries for PRIMARY as STRING. If it still has not gotten any data, it pastes whatever is in the Cut Buffer 0.

Lotus 1-2-3 Version 1.2

The Motif interface for 1-2-3 has the standard pulldown Edit menu with entries for Cut, Copy, and Paste. You select a range of cells to cut or copy by dragging the mouse with Btn1 down.

Executing Cut or Copy causes 1-2-3 to assert ownership of both the PRIMARY and CLIPBOARD selections. It will convert either selection to STRING, DELETE, LENGTH, TIMESTAMP, or TARGETS. When you cut or copy a multi-cell range, each rows are separated by vertical tabs; cells within a row are separated by newlines.

When you execute the Paste menu item, 1-2-3 first requests the CLIPBOARD selection as STRING. If that fails, it then requests the PRIMARY selection as STRING. The data is inserted starting at the current cell.

Lucid Emacs

Lucid Emacs uses both the PRIMARY and CLIPBOARD selections. Selecting text with Btn1 (either by dragging or double-clicking) makes the selected range the PRIMARY selection. This range can then be copied or cut to the CLIPBOARD selection with the **Cut** and **Copy** entries in the **Edit** pull-down menu.

As selection owner, Lucid Emacs will convert the selection to the following targets: TEXT, STRING, TARGETS, LENGTH, DELETE, FILE_NAME, CHARACTER_POSITION, LINE_NUMBER, COLUMN_NUMBER,OWNER_OS, HOST_NAME, USER, CLASS, NAME, ATOM, and INTEGER. A user can add further entries to this list via Lisp code. By default, when asking for the value of a selection, Lucid Emacs specifies a target of TEXT.

For compatibility with older programs, when a range of text is selected, Lucid Emacs copies that text into Cut Buffer 0.

R5 xterm

The R5 version of xterm allows you to select a range of text with Btn1 (by dragging, double-clicking, or triple-clicking). This range is made the PRIMARY selection and is available as STRING, TEXT, COMPOUND_TEXT, LENGTH, LIST_LENGTH, HOSTNAME, IP_ADDRESS, USER, CLASS, NAME, CLIENT_WINDOW.

Pasting is performed by clicking Btn2. If a PRIMARY selection is not available, the contents of Cut Buffer 0 are inserted.

Motif 1.1 Text Widget

The Motif Text widget uses different gestures for manipulating the PRIMARY and CLIPBOARD selections. A range of text is selected by dragging the mouse with Btn1 down. At this point, the Text widget asserts ownership of the PRIMARY selection, which it will convert to STRING, TEXT, DELETE, INSERT_SELECTION, TIMESTAMP, or TARGETS.

You can paste the PRIMARY selection by clicking Btn2. The Text widget simply requests a conversion to TEXT.

You can cut or copy the selected range of text to the CLIPBOARD selection with the **Cut** and **Copy** keys (the actual keys vary from keyboard to keyboard). The Text widget (via the Motif clipboard routines) will convert the CLIPBOARD selection to STRING, TIMESTAMP, or TARGETS.

The Paste key pastes the CLIPBOARD selection. In this case, the selection owner is asked to convert the selection to STRING.

Motif 1.2 Text Widget

The 1.2 version of the Text widget works just like the 1.1 version with the following exceptions:

It supports conversion of the PRIMARY selection to additional targets: one is the specific encoding being used (HP_ROMAN8 for instance). COMPOUND_TEXT is also supported.

When the 1.2 Text widget pastes the PRIMARY selection, it first asks for TARGETS and then asks for the richest format available (the specific encoding, COMPOUND_TEXT, or STRING).

R5 Athena Text Widget

The Athena Text widget has action routines for manipulating the PRIMARY selection. As a selection owner, it supports conversion to STRING, TEXT, COMPOUND_TEXT, LENGTH, LIST_LENGTH, CHARACTER_POSITION, DELETE (if it's not read-only), TIMESTAMP, or TARGETS.

When pasting the PRIMARY selection, the Athena Text widget just asks for conversion to STRING.

If the default translations are in effect, an attempt to paste when there is no owner of the PRIMARY selection will cause the contents of Cut Buffer 0 to be pasted.