8
Text Subwindows

This chapter describes the TEXTSW package, which allows a user or client to display and edit a sequence of ASCII characters. Figure 8-1 shows an example of a text subwindow. The text contains a vertical scrollbar but may not contain a horizontal scrollbar. The vertical scrollbar can be used to split views into several views (see Chapter 5, *Canvases and Openwin*). The font used by the text can be specified using the TEXTSW_FONT attribute, but only one font per text subwindow can be used, regardless of how many views there may be.

The contents of a text subwindow are stored in a file or in memory on the client side, not on the X server. Whether the *source* of the text is stored on disk or in memory is transparent to the user. When the user types characters in the text subwindow, the source might be changed immediately or synchronized later depending on how the text subwindow is configured. The TEXTSW package provides basic text editing features such as inserting arbitrary text into a file. It also provides complex operations such as searching for and replacing a string of text.

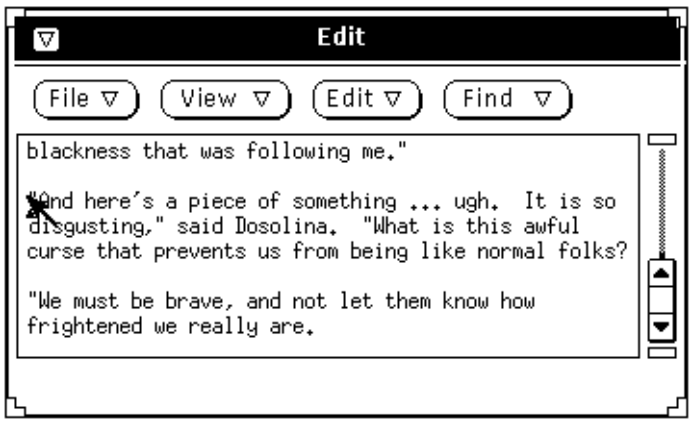

Figure 8-1. A sample text subwindow

8.1 Creating Text Subwindows

Applications need to include the file <*xview/textsw.h*> to use text subwindows. You create a text subwindow the same way you create any XView object, by calling `xv_create()` with the appropriate type parameters:

```
Textsw textsw;
textsw = (Textsw)xv_create(base_frame, TEXTSW, NULL);
```

The font used by the text can be specified using `TEXTSW_FONT`; only one font per text subwindow can be used. Figure 8-2 shows the class hierarchy for the text subwindows.

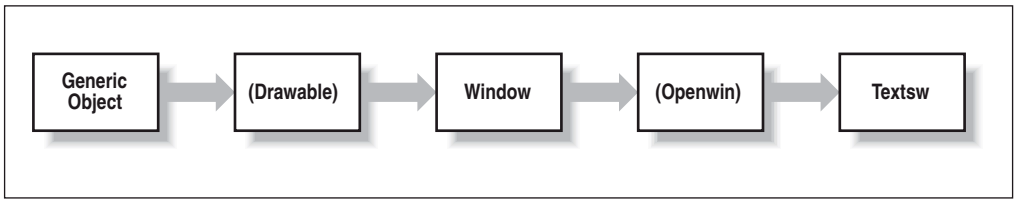

Figure 8-2. Textsw class hierarchy

8.2 Setting Text Subwindow Attributes

As for all XView objects, you can set attribute-value pairs to configure the text subwindow accordingly. Like the `CANVAS` package, the text subwindow object is subclassed from the `OPENWIN` package and can therefore be split into separate views. The package handles all of its own events and redisplaying of text, so none of these things is handled by the application.

Most text subwindow attributes are orthogonal; that is, attribute order does not effect the object. In a few cases, however, the attributes in a list might interact, so you must specify them in a particular order. Such cases are noted in the sections that follow. For example, you must pass `TEXTSW_STATUS` first in any call to `xv_create()`, if you want to find the status after setting some other attribute in the same call.

8.3 Text Subwindow Contents

The contents of a text subwindow are a sequence of characters. Each character can be uniquely identified by its position in the sequence (type `Textsw_index`). Editing operations, such as inserting and deleting text, can cause the index of successive characters to change. The valid indices are 0 through *length* –1 inclusive, where *length* is the number of characters currently in the text subwindow, returned by the `TEXTSW_LENGTH` attribute.

The text subwindow has a notion of the current index after which the next character will be inserted. This is called the *insertion point* and is indicated by a caret, as shown in Figure 8-3.

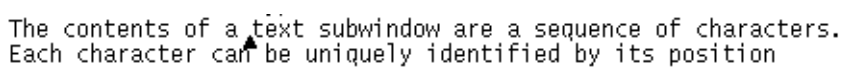

Figure 8-3. A caret marks the insertion point

8.4 Editing a Text Subwindow

A text subwindow can be edited by the user or by a client program. When you create a text subwindow, the user is normally allowed to edit it. By using the special attributes discussed in this section, the client program can edit the subwindow. These edits are then stored in */tmp/text****Process-id.Counter***.

The next five sections explain the functions and attributes that you will use to load, read, write, edit, and finally save a text file.

8.4.1 Loading a File

You can load a file into a text subwindow by using `TEXTSW_FILE`, as in:

```
xv_set(textsw, TEXTSW_FILE, file_name, NULL);
```

Keep in mind that if the existing text has been edited, then these edits will be lost. To avoid such loss, first check whether there are any outstanding edits by calling:

```
int modified = (int)xv_get(textsw, TEXTSW_MODIFIED)
```

If there have been updates, you may choose to synchronize with the *source* if necessary. That is, if the existing text is part of a file, you can overwrite the existing changes before loading in a new file.

The above call to `xv_set()`, which loads the new file, positions the new text so that the first character displayed has the same index as the first character that was displayed in the previous file. This is probably not what you want. The code segment below shows how to load the file at a set position:

```
xv_set(textsw,
    TEXTSW_FILE,  file_name,
    TEXTSW_FIRST, position,
    NULL);
```

The first character displayed has its index set by `position`. The order of these attributes matters. Because attributes are evaluated in the order given, reversing the order would first reposition the existing file, then load the new file. This would cause an unnecessary repaint. It would also mis-position the old file if it was shorter than `position`.

8.4.2 Checking the Status of the Text Subwindow

Both of the calls in the previous example blindly trust that the load of the new file was successful. This is, in general, a bad idea. The following code segment shows how to find out whether the load succeeded, and if not, why it failed:

```
Textsw textsw;
Textsw_status status;

textsw = (Textsw)xv_create(base_frame,
    TEXTSW,
    TEXTSW_STATUS,   &status,
    TEXTSW_FILE,     file_name,
    TEXTSW_FIRST,    position,
    NULL);
```

NOTE

The TEXTSW_STATUS attribute and handle must appear in the attribute list *before* the operation whose status you want to determine.

The TEXTSW_STATUS attribute is only valid for `xv_create()`.

The range of values for such a variable are enumerated in Table 8-1. Note that in the first column, each value begins with the prefix TEXTSW_STATUS_, which has been omitted from the table to improve readability.

Table 8-1. Range of Values for Status Variables

Value (TEXTSW_STATUS_ ...)	Description
OKAY	The operation encountered no problems.
BAD_ATTR	The attribute list contained an illegal or unrecognized attribute.
BAD_ATTR_VALUE	The attribute list contained an illegal value for an attribute, usually an out-of-range value for an enumeration.
CANNOT_ALLOCATE	A call to `calloc`(2) or `malloc`(2) failed.
CANNOT_OPEN_INPUT	The specified input file does not exist or cannot be accessed.
CANNOT_INSERT_FROM_FILE	The operation encountered a problem when trying to insert from file.
OUT_OF_MEMORY	The operation ran out of memory while editing in memory.
OTHER_ERROR	The operation encountered a problem not covered by any of the other error indications.

8.4.3 Writing to a Text Subwindow

To insert text into a text subwindow at the current insertion point, call:

```
Textsw_index
textsw_insert(textsw, buf, buf_len)
    Textsw  textsw;
    char   *buf;
    int     buf_len;
```

The return value is the number of characters actually inserted into the text subwindow. This number will equal `buf_len` unless either the text subwindow has had a memory allocation failure or the portion of text containing the insertion point is read only. The insertion point is moved forward by the number of characters inserted.

This routine does not do terminal-style interpretation of the input characters. Thus, editing characters (such as CTRL-H or DEL for character erase) are simply inserted into the text subwindow rather than performing edits to the existing contents of the text subwindow. To emulate a terminal, scan the characters to be inserted and invoke `textsw_edit()` where appropriate, as described in the next section.

8.4.3.1 Setting the insertion point

The attribute `TEXTSW_INSERTION_POINT` is used to interrogate and set the insertion point. For instance, the following call determines where the insertion point is:

```
Textsw_index point;

point = (Textsw_index)xv_get(textsw, TEXTSW_INSERTION_POINT);
```

Whereas the following call sets the insertion point to be just before the third character of the text:

```
xv_set(textsw, TEXTSW_INSERTION_POINT, 2, NULL);
```

To set the insertion point at the end of the text, set `TEXTSW_INSERTION_POINT` to the special index `TEXTSW_INFINITY`. This call does not ensure that the new insertion point will be visible in the text subwindow, even if `TEXTSW_INSERT_MAKES_VISIBLE` is `TRUE`. To guarantee that the caret will be visible afterwards, call `textsw_possibly_normalize()`, a procedure that is described later in this chapter.

8.4.4 Reading from a Text Subwindow

Many applications that incorporate text subwindows never need to read the contents of the text directly from the text subwindow. For instance, the text subwindow might display text for the user to view but not to edit.

Even when the user is allowed to edit text, some applications simply wait for the user to perform some action that indicates that all of the edits have been made. The application can then use either `textsw_save()` or `textsw_store_file()` to place the text in the file.

The text can then be read via the usual file input utilities, or the file itself can be passed off to another routine or program.

It is, however, useful to be able to directly examine the text in the text subwindow. You can do this using the TEXTSW_CONTENTS attribute. The code fragment below illustrates how to use TEXTSW_CONTENTS to get a span of characters from the text subwindow. It gets 1000 characters beginning at position 500 out of the text subwindow and places them into a NULL-terminated string.

```
#define TO_READ 1000

char         buf[TO_READ+1];
Textsw_index next_pos;

next_pos = (Textsw_index) xv_get(textsw, TEXTSW_CONTENTS, 500,
                                 buf, TO_READ);

if (next_pos != 500+TO_READ) {
    /* handle error case */
} else
    buf[TO_READ] = '\0';
```

8.4.5 Deleting Text

You can delete a contiguous span of characters from a text subwindow by calling:

```
Textsw_index
textsw_delete(textsw, first, last_plus_one)
    Textsw        textsw;
    Textsw_index  first, last_plus_one;
```

`first` specifies the first character of the span that will be deleted; `last_plus_one` specifies the first character *after* the span that will *not* be deleted. `first` should be less than or equal to `last_plus_one`. To delete to the end of the text, pass the special value TEXTSW_INFINITY for `last_plus_one`.

The return value is the number of characters deleted or:

```
last_plus_one - first
```

unless the specified span is read-only. If the insertion point is in the span being deleted, it will be left at `first`.

A side effect of calling `textsw_delete()` is that the deleted characters become the contents of the global Clipboard. To remove the characters from the text subwindow without affecting the Clipboard, call:

```
Textsw_index
textsw_erase(textsw, first, last_plus_one)
    Textsw        textsw;
    Textsw_index  first, last_plus_one;
```

Again, the return value is the number of characters removed, and `last_plus_one` can be TEXTSW_INFINITY.

Both of these procedures will return 0 if the operation fails.

8.4.6 Emulating an Editing Character

You can emulate the behavior of an editing character, such as CTRL-H, with `textsw_edit()`:

```
Textsw_index
textsw_edit(textsw, unit, count, direction)
    Textsw    textsw;
    unsigned  unit, count, direction;
```

Depending on the value of `unit`, this routine will erase either a character, a word, or a line. Set `unit` to:

- `TEXTSW_UNIT_IS_CHAR` to erase individual characters.
- `TEXTSW_UNIT_IS_WORD` to erase the span of characters that make up a word (including any intervening white space or other nonword characters).
- `TEXTSW_UNIT_IS_LINE` to erase all characters in the line on one side of the insertion point.

If the `direction` parameter is 0, the operation will affect characters after the insertion point; otherwise, it will affect characters before the insertion point.

The `count` parameter determines the number of times the operation will be applied. Set it to 1 to do the edit once or to a value greater than 1 to do multiple edits in a single call. `textsw_edit()` returns the number of characters actually removed.

For example, suppose you want to interpret the function key `F7` as meaning *delete word forward*. On receiving the event code for the `F7` key, you would make the call:

```
textsw_edit(textsw, TEXTSW_UNIT_IS_WORD, 1, NULL);
```

8.4.7 Replacing Characters

While a span of characters can be replaced by calling `textsw_erase()` followed by `textsw_insert()`, character replacement is done most efficiently by calling:

```
Textsw_index
textsw_replace_bytes(textsw, first, last_plus_one, buf, buf_len)
    Textsw        textsw;
    Textsw_index  first, last_plus_one;
    char         *buf;
    int           buf_len;
```

The span of characters to be replaced is specified by `first` and `last_plus_one`, just as in the call to `textsw_erase()`. The new characters are specified by `buf` and `buf_len`, just as in the call to `textsw_insert()`. Once again, if `last_plus_one` is `TEXTSW_INFINITY`, the replace operation affects all characters from `first` to the end of the text. If the insertion point is in the span being replaced, it will be left at:

```
first + buf_len
```

The return value is the net number of bytes inserted. The number is negative if the original string is longer than the one that replaces it. If a problem occurs when an attempt is made to replace a span, it will return an error code of 0.

`textsw_replace_bytes()`, like `textsw_erase()`, does *not* put the characters it removes on the global Clipboard.

8.4.8 The Editing Log

All text subwindows allow the user to undo editing actions. The `TEXTSW` package keeps a running log of all the edits. If a file is associated with the text subwindow, this log is kept in a file in the */tmp* directory. This file can grow until the file system in which this directory resides runs out of space. To limit the size of the edit log and to avoid filling up all of */tmp*, the user can set the text wrap-around size with `TEXTSW_WRAPAROUND_SIZE`. If there is no associated file, the edit log is kept in memory, and the maximum size of the log is controlled by the attribute `TEXTSW_MEMORY_MAXIMUM`, which defaults to 20,000 bytes.

Unfortunately, once a memory-resident edit log has reached its maximum size, no more characters can be inserted into or removed from the text subwindow. In particular, since deletions as well as insertions are logged, space cannot be recovered by deleting characters.

It is important to understand how the edit log works, since you might want to use a text subwindow with no associated file to implement a temporary scratch area or error message log. If such a text subwindow is used for a long time, the default limit of 20,000 bytes might well be reached, and either the user or your code will be unable to insert any more characters, even though only a few characters might be visible in the text subwindow. Therefore, it is recommended to set `TEXTSW_MEMORY_MAXIMUM` to a much higher value, say 200,000.

8.4.9 Which File is Being Edited?

To find out the name of the file in the text subwindow, call:

```
int
textsw_append_file_name(textsw, name)
    Textsw  textsw;
    char   *name;
```

If the text subwindow is editing memory, then this routine will return a nonzero value. Otherwise, it will return 0 and append the name of the file to the end of `name`. The following code gets the name of the current file:

```
char name[BUFSIZ];

name[0] = '\0';
if (textsw_append_file_name(textsw, name) == 0)
    printf("File name is: %s\n", name);
```

8.4.9.1 Interactions with the file system

Suppose the current file is called *myfile*. If the user chooses `textsw_save()`), the following sequence of file operations occurs:

- *myfile* is copied to *myfile%*.
- The contents of *myfile%* are combined with information from the edit log file (*/tmp/Text**Process-id.Counter***) and written over *myfile*, thereby preserving all its permissions, etc.
- The edit log file is removed from */tmp*.

If *myfile* is a symbolic link to *../some_dir/otherfile*, then the backup file is created as *../some_dir/otherfile%*.

Keep in mind that the user can change the current directory by selecting "Load File" or "Set Directory" from the text subwindow menu. If *myfile* is a relative path name, then both the copy to *myfile%* and the save take place in the current directory.

8.5 Saving Edits in a Subwindow

To save any edits made to a file currently loaded into a text subwindow call:

```
unsigned
textsw_save(textsw, locx, locy)
    Textsw  textsw;
    int     locx, locy;
```

`locx` and `locy` are relative to the upper-left corner of the text subwindow and are used to position the upper-left corner of the alert should the save fail for some reason—usually they should be 0. The return value is 0 if and only if the save succeeded.

8.5.1 Storing Edits

The text subwindow might not contain a file, or the client might wish to place the edited version of the text (whether or not the original text came from a file) in some specific file. To store the contents of a text subwindow to a file, call:

```
unsigned
textsw_store_file(textsw, filename, locx, locy)
    Textsw  textsw;
    char   *filename;
    int     locx, locy;
```

Again, `locx` and `locy` are used to position the upper-left corner of the message box. The return value is 0 if and only if the store succeeded.

By default, this call changes the file that the text subwindow is editing, so that subsequent saves will save the edits to the new file. To override this policy, set the attribute `TEXTSW_STORE_CHANGES_FILE` to `FALSE`.

8.5.2 Discarding Edits

To discard the edits performed on the contents of a text subwindow, call:

```
void
textsw_reset(textsw, locx, locy)
    Textsw  textsw;
    int     locx, locy;
```

`locx` and `locy` are as above. Note that if the text subwindow contains a file that has not been edited, the effect of `textsw_reset` is to unload the file and replace it by memory provided by the `TEXTSW` package; thus, the user will see an absolutely empty text subwindow. Alternatively, if the text subwindow was already editing memory, then another, untouched, piece of primary memory will be provided and the edited piece will be deallocated.

8.6 Setting the Contents of a Text Subwindow

The rest of this chapter describes the other functions that are available for text subwindows. These features include setting the contents of a subwindow, setting the primary selection, and dealing with multiple or split views.

You might want to set the initial contents of a text subwindow that your application uses. To set the initial contents of a text subwindow, use one of three attributes: `TEXTSW_INSERT_FROM_FILE`, `TEXTSW_FILE_CONTENTS`, and `TEXTSW_CONTENTS`. Each attribute is illustrated in code fragments given below.

8.6.1 TEXTSW_FILE_CONTENTS

The attribute `TEXTSW_FILE_CONTENTS` allows a client to initialize the text subwindow contents from a file yet still edit the contents in memory. The user can return a text subwindow to its initial state after an editing session by choosing “Undo All Edits” in the text menu.

The code fragment below shows how you would use this attribute:

```
extern char *filename;

xv_set(textsw,
   TEXTSW_FILE_CONTENTS, filename,
   TEXTSW_FIRST,         0,
   NULL);
```

When the client calls the undo routine and `filename` is not a null string, the memory used by the text subwindow is reinitialized with the contents of the file specified by `filename`.

When the client calls the undo routine and the `filename` is a null string, the memory used by the text subwindow is reinitialized with the previous contents of the text subwindow.

8.6.2 TEXTSW_CONTENTS

TEXTSW_CONTENTS lets you insert a text string from memory, instead of a file, into the text subwindow. The default for this attribute is NULL.

Using xv_create() with this attribute specifies the initial contents for a nonfile text subwindow.

Using xv_set() with this attribute sets the contents of a window, as in:

```
xv_set(textsw, TEXTSW_CONTENTS, "text", NULL);
```

If you use xv_get() with this attribute, you will need to provide additional parameters, as in:

```
xv_get(textsw, TEXTSW_CONTENTS, pos, buf, buf_len);
```

The return value is the next position to be read. The buffer array:

```
buf[0 ... buf_len-1]
```

is filled with the characters from textsw beginning at the index pos and is NULL-terminated only if there were too few characters to fill the buffer.

8.6.3 TEXTSW_INSERT_FROM_FILE

TEXTSW_INSERT_FROM_FILE allows a client to insert the contents of a file into a text subwindow at the current insertion point. It is the programming equivalent of a user choosing “Include File” from the text menu.

The code below demonstrates this attribute:

```
Textsw          textsw;
Textsw_status   status;

xv_set(textsw,
    TEXTSW_STATUS,           &status,
    TEXTSW_INSERT_FROM_FILE, filename,
    NULL);
```

Three status values can be returned for this attribute when the argument TEXTSW_STATUS is passed in the same call to xv_create() or xv_set():

```
TEXTSW_STATUS_OKAY
TEXTSW_STATUS_CANNOT_INSERT_FROM_FILE
TEXTSW_STATUS_OUT_OF_MEMORY
```

8.7 Positioning the Text Displayed in a Text Subwindow

Usually, more text is managed by the text subwindow than can be displayed all at once. As a result, it is often necessary to determine the indices of the characters that are being displayed and to control exactly which portion of the text is visible.

8.7.1 Screen Lines and File Lines

When there are long lines in the text, it is necessary to distinguish between two definitions of "line of text."

A *screen line* reflects what is actually displayed on the screen. A line begins with the leftmost character in the subwindow and continues across until either a newline character or the right edge of the subwindow is encountered. A *file line*, on the other hand, can only be terminated by the newline character. It is defined as the span of characters starting after a newline character (or the beginning of the file) running through the next newline character (or the end of the file).

Whenever the right edge of the subwindow is encountered before the newline, if the following attribute-value pair were specified:

```
TEXTSW_LINE_BREAK_ACTION, TEXTSW_WRAP_AT_CHAR
```

then the next character and its successors would be displayed on the next lower screen line. In this case, there would be two screen lines, but only one file line. From the perspective of the display there are two lines; from the perspective of the file, only one. On the other hand, if the following attribute-value pair were specified:

```
TEXTSW_LINE_BREAK_ACTION, TEXTSW_WRAP_AT_WORD
```

then the entire word would be displayed on the next line.

Unless otherwise specified, all text subwindow attributes and procedures use the *file line* definition. Line indices have a zero-origin, like the character indices; that is, the first line has index 0, not 1.

8.7.2 Absolute Positioning

Two attributes are provided to allow you to specify which portion of the text is displayed in the text subwindow.

Setting the attribute `TEXTSW_FIRST` to a given index causes the first character of the line containing the index to become the first character displayed in the text subwindow. Thus, the following call causes the text to be positioned so that the first displayed character is the first character of the line that contains index 1000:

```
xv_set(textsw, TEXTSW_FIRST, 1000, NULL);
```

Since the text subwindow is subclassed from the `OPENWIN` package and can be split into several views, the previous code fragment would only cause the positioning of one view. To

position all of the views in a text subwindow, use the attribute TEXTSW_FOR_ALL_VIEWS, as in the following call:

```
xv_set(textsw,
    TEXTSW_FOR_ALL_VIEWS, TRUE,
    TEXTSW_FIRST,         1000,
    NULL);
```

Conversely, the following call retrieves the index of the first displayed character:

```
index = (Textsw_index)xv_get(textsw, TEXTSW_FIRST);
```

A related attribute, useful in similar situations, is TEXTSW_FIRST_LINE. When used in a call on xv_set() or xv_get(), the value is a file line index within the text.

You can determine the character index that corresponds to a given line index (both zero-origin) within the text by calling:

```
Textsw_index
textsw_index_for_file_line(textsw, line)
    Textsw  textsw;
    int     line;
```

The return value is the character index for the first character in the line, so character index 0 always corresponds to line index 0.

8.7.3 Relative Positioning

To move the text in a text subwindow up or down by a small number of lines, call the routine:

```
void
textsw_scroll_lines(textsw, count)
Textsw  textsw;
int     count;
```

A positive value for count causes the text to scroll up, while a negative value causes the text to scroll down.

When calling textsw_scroll_lines(), you might want to know how many screen lines are in the text subwindow. You can find this out by calling:

```
int
textsw_screen_line_count(textsw)
    Textsw  textsw;
```

8.7.4 Which File Lines are Visible?

Exactly which file lines are visible on the screen is determined by calling:

```
void
textsw_file_lines_visible(textsw, top, bottom)
    Textsw  textsw;
    int    *top, *bottom;
```

This routine fills in the addressed integers with the file line indices of the first and last file lines being displayed in the specified text subwindow.

8.7.4.1 Guaranteeing what is visible

To ensure that a particular line or character is visible, call:

```
void
textsw_possibly_normalize(textsw, position)
    Textsw       textsw;
    Textsw_index position;
```

The text subwindow must be displayed on the screen before this function will work.

If the character at the specified `position` is already visible, then this routine does nothing. If it is not visible, then it repositions the text so that it is visible and at the top of the subwindow.

If a particular character should always be at the top of the subwindow, then calling the following routine is more appropriate:

```
void
textsw_normalize_view(textsw, position)
    Textsw       textsw;
    Textsw_index position;
```

8.7.4.2 Ensuring that the insertion point is visible

Most of the programmatic editing actions do not update the text subwindow to display the caret, even if `TEXTSW_INSERT_MAKES_VISIBLE` is set. If you want to ensure that the insertion point is visible, use:

```
textsw_possibly_normalize(textsw,
    (Textsw_index) xv_get(textsw, TEXTSW_INSERTION_POINT));
```

8.8 Finding and Matching a Pattern

A common operation performed on text is to find a span of characters that match some specification. The text subwindow provides several rudimentary pattern matching facilities. This section describes two functions that you can call in order to perform similar operations.

8.8.1 Matching a Span of Characters

To find the nearest span of characters that match a pattern, call:

```
int
textsw_find_bytes(textsw, first, last_plus_one, buf,
                  buf_len, flags)
```

```
Textsw        textsw;
Textsw_index *first, *last_plus_one;
char         *buf;
unsigned      buf_len;
unsigned      flags;
```

The pattern to match is specified by `buf` and `buf_len`. The matching operation looks for an exact and literal match—it is sensitive to case and does not recognize any kind of meta-character in the pattern. `first` specifies the position at which to start the search. If `flags` is 0, the search proceeds forward through the text; if `flags` is 1, the search proceeds backwards. The return value is –1 if the pattern cannot be found; otherwise it is some non-negative value, in which case the indices addressed by `first` and `last_plus_one` will have been updated to indicate the span of characters that match the pattern.

8.8.2 Matching a Specific Pattern

Another useful operation is to find delimited text. For example, you might want to find the starting and ending brace in a piece of code. To find a matching pattern, call:

```
int
textsw_match_bytes(textsw, first, last_plus_one,
                   start_sym, start_sym_len,
                   end_sym, end_sym_len, field_flag)
Textsw        textsw;
Textsw_index *first, *last_plus_one;
char         *start_sym, *end_sym;
int           start_sym_len, end_sym_len;
unsigned      field_flag;
```

`first` stores the starting position of the pattern that you want to search for. `last_plus_one` stores the cursor position of the end pattern. Its value is one position past the text. `start_sym` and `end_sym` store the beginning position and ending position of the pattern, respectively. `start_sym_len` and `end_sym_len` store the starting and ending pattern's length, respectively.

Use one of the following three field flag values to search for matches:

`TEXTSW_DELIMITER_FORWARD`
: Begins from `first` and searches forward until it finds `start_sym` and matches it forward with `end_sym`.

`TEXTSW_DELIMITER_BACKWARD`
: Begins from `first` and searches backward for `end_sym` and matches it backward with `start_sym`.

`TEXTSW_DELIMITER_ENCLOSE`
: Begins from `first` and expands both directions to match `start_sym` and `end_sym` of the next level.

If no match is found, then `textsw_match_bytes()` will return a value of –1. If a match is found, then it will return the index of the first match.

The code fragment below can be used to find delimited text. Notice that the `field_flag` value is `TEXTSW_DELIMITER_FORWARD`.

```
Textsw_index     first, last_plus_one, pos;

first = (Textsw_index) xv_get(textsw, TEXTSW_INSERTION_POINT);
pos = textsw_match_bytes(textsw, &first, &last_plus_one,
                 "/*", 2,
                 "*/", 2, TEXTSW_DELIMITER_FORWARD);
if (pos > 0) {
     textsw_set_selection(textsw, first, last_plus_one, 1);
     xv_set(textsw, TEXTSW_INSERTION_POINT, last_plus_one, NULL);
} else
    (void) window_bell(textsw);
```

This code searches forward from `first` until it finds the starting /* and matches it forward with the next */. If no match is found, a bell will ring in the text subwindow.

8.9 Marking Positions

Often a client wants to keep track of a particular character or group of characters that are in the text subwindow. Given that arbitrary editing can occur in a text subwindow and that it is very tedious to intercept and track all of the editing operations applied to a text subwindow, it is often easier to simply place one or more marks at various positions in the text subwindow. These marks are automatically updated by the text subwindow to account for user and client edits. There is no limit to the number of marks you can add.

A new mark is created by calling:

```
Textsw_mark
textsw_add_mark(textsw, position, flags)
    Textsw        textsw;
    Textsw_index  position;
    unsigned      flags;
```

The `flags` argument is either `TEXTSW_MARK_DEFAULTS` or `TEXTSW_MARK_MOVE_AT_INSERT`. The latter causes an insertion at the marked position to move the mark to the end of the inserted text, whereas the former causes the mark to not move when text is inserted at the mark's current position. As an example, suppose that the text managed by the text subwindow consists of the two lines:

```
this is the first line
not this, which is the second
```

Assume a mark is set at position 5 (just before the *i* in *is* on the first line) with `flags` of `TEXTSW_MARK_MOVE_AT_INSERT`.

If the user makes a selection just before the *is* (thereby placing the insertion point before the *i*, at position 5) and types an *h*, making the text read:

```
this his the first line
not this, which is the second
```

the mark moves with the insertion point and they both end up at position 6.

However, if the `flags` had been `TEXTSW_MARK_DEFAULTS`, then the mark would remain at position 5 after the user typed the *h*, although the insertion point moved on to position 6.

Now, suppose instead that the user made a selection before the *this* on the first line, and typed *Kep*, making the text read:

```
Kepthis is the first line
not this, which is the second
```

In this case, no matter what `flags` the mark had been created with, it would end up at position 8, still just before the *i* in *is*.

If a mark is in the middle of a span of characters that is subsequently deleted, the mark moves to the beginning of the span. Going back to the original scenario, with the original text and the mark set at position 5, assume that the user deletes from the *h* in *this* through the *e* in *the* on the first line, resulting in the text:

```
te first line
not this, which is the second
```

When the user is done, the mark will be at position 1, just before the *e* in *te*.

The current position of a mark is determined by calling:

```
Textsw_index
textsw_find_mark(textsw, mark)
    Textsw       textsw;
    Textsw_mark  mark;
```

An existing mark is removed by calling:

```
void
textsw_remove_mark(textsw, mark)
    Textsw       textsw;
    Textsw_mark  mark;
```

Note that marks are dynamically allocated, and it is the client's responsibility to keep track of them and to remove them when they are no longer needed.

8.9.1 Getting a Text Selection

A user selects a portion of the contents of the text subwindow using a pointer. A text selection is indicated on the screen with reverse-video highlighting. An application needs to know which window has the current selection and what the contents of a text selection are. The `TEXTSW` package does not provide procedures to get this information. Instead, these functions are carried out by the Selection Service. For an example of how this is done, see Chapter 18, *Selections*. Figure 8-4 shows a text selection.

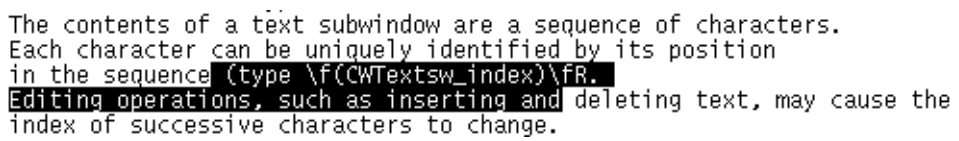

Figure 8-4. A text selection

8.9.2 Setting the Text Selection

Primary and secondary selections are maintained. The primary or secondary selection can be set by calling the following:

```
void
textsw_set_selection(textsw, first, last_plus_one, type)
    Textsw        textsw;
    Textsw_index  first, last_plus_one;
    unsigned      type;
```

A value of 1 for type means *primary selection*, while a value of 2 means *secondary selection* and a value of 17 is *pending delete*. Note that there is no requirement that all or part of the selection be visible; use `textsw_possibly_normalize()` to guarantee visibility (see Section 8.7.4, “Which File Lines are Visible?”).

8.10 Dealing with Multiple Views

By splitting a text view, the user can create multiple views of the text being managed by the text subwindow. Although these additional views are usually transparent to the client code controlling the text subwindow, it might occasionally be necessary for a client to deal directly with all of the views. This is accomplished by using the following routines, with the knowledge that split views are simply extra text subwindows that happen to share the text of the original text subwindow.

```
Textsw
textsw_first(textsw)
    Textsw textsw;
```

Given an arbitrary view out of a set of multiple views, `textsw_first()` returns the first view (currently, this is the original text subwindow that the client created). To move through the other views of the set, call:

```
Textsw
textsw_next(textsw)
    Textsw textsw;
```

Given any view of the set, `textsw_next()` returns some other member of the set or `NULL` if there are none left to enumerate. The loop coded below is guaranteed to process all of the views in the set:

```
for (textsw = textsw_first(any_split); textsw;
             textsw = textsw_next(textsw)) {
    /* processing involving textsw */
}
```

When you create a text subwindow, take into account that the user might split the window. If you try to do something like enlarge the window, you might run into problems.

8.11 Text Subwindow Destroy Confirmation

A confirmation notice is displayed when a text subwindow is about to be destroyed. A text subwindow is destroyed when the text subwindow or its enclosing frame is the object of a `xv_destroy()` call (this may occur when application is quit from the window manager's menu). Supplying the text subwindow confirmation notice is referred to as *vetoing the destroy*. A confirmation notice is provided when the text subwindow's *ignore limit* has been reached. The ignore limit specifies the number of edits permitted before the confirmation notice is displayed and is set with `TEXTSW_IGNORE_LIMIT`. Valid values for `TEXTSW_IGNORE_LIMIT` are 0, meaning destroy will be vetoed if any edits have been done, and `TEXTSW_INFINITY`, meaning the destroy will never be vetoed.

8.12 Notifications from a Text Subwindow

The text subwindow notifies its client about interesting changes in the subwindow's or text's state by calling a notification procedure. It also calls this procedure in response to user actions. If the client does not provide an explicit notification procedure by using the attribute `TEXTSW_NOTIFY_PROC`, then the text subwindow provides a default procedure. The declaration for this procedure looks like:

```
void
notify_proc(textsw, avlist)
    Textsw      textsw;
    Attr_avlist  avlist;
```

`avlist` contains attributes that are the members of the `Textsw_action` enumeration.

Your notification procedure must be careful either to process all of the possible attributes or to pass through the attributes that it does not process to the standard notification procedure. This is important because among the attributes that can be in the *avlist* are those that cause the standard notification procedure to implement the possible *Front*, *Back*, *Open*, *Close*, and *Quit* accelerators of the user interface.

Example 8-1 presents a client notify procedure for a text subwindow.

Example 8-1. Client notify procedure for a text subwindow

```
void (*textsw_default_notify)();

void
client_notify_proc(textsw, attributes)
```

Example 8-1. Client notify procedure for a text subwindow (continued)

```
Textsw         textsw;
Attr_avlist    attributes;
{
    int  pass_on   = FALSE;
    Attr_avlist    attrs;

    for (attrs = attributes; *attrs; attrs = attr_next(attrs)) {
        switch ((Textsw_action)(*attrs)) {
            case TEXTSW_ACTION_CAPS_LOCK:
                /* Swallow this attribute */
                ATTR_CONSUME(*attrs);
                break;
            case TEXTSW_ACTION_CHANGED_DIRECTORY:
                /* Monitor the attribute, don't swallow it */
                strcpy(current_directory, (char *)attrs[1]);
                pass_on = TRUE;
                break;
            default:
                pass_on = TRUE;
                break;
        }
    }
    if (pass_on)
        textsw_default_notify(textsw, attributes);
}

textsw_default_notify =
    (void (*)())xv_get(textsw, TEXTSW_NOTIFY_PROC);
xv_set(textsw, TEXTSW_NOTIFY_PROC, client_notify_proc, NULL);
```

The `Textsw_action` attributes that can be passed to your notify procedure are listed in Table 8-2. Note that in the first column, each attribute begins with the prefix `TEXTSW_ACTION_`, which has been omitted from the table to improve readability. Remember that the attributes constitute a special class that are passed to your text subwindow notification procedure. They are not attributes of the text subwindow in the usual sense and cannot be retrieved or modified using `xv_get()` or `xv_set()`.

Table 8-2. Textsw_action Attributes

Attribute (`TEXTSW_ACTION_ ...`)	Type	Description
`CAPS_LOCK`	`Boolean`	The user pressed the Caps Lock key to change the setting of the Caps Lock (it is initially 0, meaning off).
`CHANGED_DIRECTORY`	`char *`	The current working directory for the process has been changed to the directory named by the provided string value.
`EDITED_FILE`	`char *`	The file named by the provided string value has been edited. Appears once per session of edits (see below).

Table 8-2. Textsw_action Attributes (continued)

Attribute (TEXTSW_ACTION_ ...)	Type	Description
EDITED_MEMORY	(no value)	Monitors whether an empty text subwindow has been edited.
FILE_IS_READONLY	char *	The file named by the provided string value does not have write permission.
LOADED_FILE	char *	The text subwindow is being used to view the file named by the provided string value.
TOOL_CLOSE	(no value)	The frame containing the text subwindow should become iconic.
TOOL_DESTROY	Event *	The tool containing the text subwindow should exit, without checking for a veto from other subwindows. The value is the user action that caused the destroy.
TOOL_QUIT	Event *	The tool containing the text subwindow should exit normally. The value is the user action that caused the exit.
TOOL_MGR	Event *	The tool containing the text subwindow should do the window manager operation associated with the provided event value.
USING_MEMORY	(no value)	The text subwindow is being used to edit a string stored in primary memory, not a file.

The attribute TEXTSW_ACTION_EDITED_FILE is a slight misnomer, as it is given to the notify procedure *after* the first edit to *any* text, whether or not it came from a file. This notification happens only once per session of edits, whereas, on the other hand, notification of TEXTSW_ACTION_LOADED_FILE is considered to terminate the old session and start a new one.

NOTE

The attribute TEXTSW_ACTION_LOADED_FILE must be treated very carefully because the notify procedure gets called with this attribute in several situations: after a file is initially loaded, after any successful "Save Current File" menu operation, after an "Undo All Edits" menu operation, and during successful calls to textsw_reset(), textsw_save(), and textsw_store().

The appropriate response by the procedure is to interpret these notifications as being equivalent to:

> *The text subwindow is displaying the file named by the provided string value; no edits have been performed on the file yet. In addition, any previously displayed or edited file has been either reset, saved, or stored under another name*.

8.12.1 Text Subwindow Interposition

If you need to interpose on a text subwindow, get the text subwindow's *view window* and interpose on it.

```
Xv_Window window;
Textsw    textsw;
int       win_no;

window = (Xv_Window)xv_get(textsw, OPENWIN_NTH_VIEW, win_no);
```

For more information on registering events for text subwindows, see Section 6.3, "Registering Events;" for more information on interposition, refer to Section 20.9, "Interposition."

8.13 Text Subwindow Package Summary

Table 8-3 lists the procedures and macros for the `TEXTSW`. Table 8-4 lists the attributes for the `TEXTSW` package. This information is described fully in the *XView Reference Manual*.

Table 8-3. Text Subwindow Procedures and Macros

Procedures and Macros	
`textsw_add_mark()`	`textsw_next()`
`textsw_append_file_name()`	`textsw_normalize_view()`
`textsw_delete()`	`textsw_notify_proc()`
`textsw_edit()`	`textsw_possibly_normalize()`
`textsw_erase()`	`textsw_remove_mark()`
`textsw_file_lines_visible()`	`textsw_replace_bytes()`
`textsw_find_bytes()`	`textsw_reset()`
`textsw_find_mark()`	`textsw_save()`
`textsw_first()`	`textsw_screen_line_count()`
`textsw_index_for_file_line()`	`textsw_scroll_lines()`
`textsw_insert()`	`textsw_set_selection()`
`textsw_match_bytes()`	`textsw_store_file()`

Table 8-4. Text Subwindow Attributes

TEXTSW_AGAIN_RECORDING	TEXTSW_IGNORE_LIMIT
TEXTSW_AUTO_INDENT	TEXTSW_INSERT_FROM_FILE
TEXTSW_AUTO_SCROLL_BY	TEXTSW_INSERT_MAKES_VISIBLE
TEXTSW_BLINK_CARET	TEXTSW_INSERTION_POINT
TEXTSW_BROWSING	TEXTSW_LENGTH
TEXTSW_CHECKPOINT_FREQUENCY	TEXTSW_LINE_BREAK_ACTION
TEXTSW_CLIENT_DATA	TEXTSW_LOWER_CONTEXT
TEXTSW_CONFIRM_OVERWRITE	TEXTSW_MEMORY_MAXIMUM
TEXTSW_CONTENTS	TEXTSW_MULTI_CLICK_SPACE
TEXTSW_CONTROL_CHARS_USE_FONT	TEXTSW_MULTI_CLICK_TIMEOUT
TEXTSW_DESTROY_VIEW	TEXTSW_NOTIFY_PROC
TEXTSW_DISABLE_CD	TEXTSW_READ_ONLY
TEXTSW_DISABLE_LOAD	TEXTSW_STATUS
TEXTSW_EDIT_COUNT	TEXTSW_STORE_CHANGES_FILE
TEXTSW_EXTRAS_CMD_MENU	TEXTSW_SUBMENU_EDIT
TEXTSW_FILE	TEXTSW_MODIFIED
TEXTSW_FILE_CONTENTS	TEXTSW_SUBMENU_FILE
TEXTSW_FIRST	TEXTSW_SUBMENU_FIND
TEXTSW_FIRST_LINE	TEXTSW_SUBMENU_VIEW
TEXTSW_FONT	TEXTSW_UPPER_CONTEXT
TEXTSW_HISTORY_LIMIT	
XV_LEFT_MARGIN	XV_RIGHT_MARGIN