3
Procedures and Macros

This section lists the XView procedures and macros in alphabetical order. For each procedure or macro, an explanation of its use is provided. Following each procedure is a synopsis that includes the procedure's parameters, if there are any. The parameters are described after the synopsis.

attr_create_list()

Takes an attribute-value list and converts it into an array of `Attr_attributes`, also known as an `Attr_avlist`. `ATTRIBUTE_STANDARD_SIZE`, defined in *<xview/attr.h>*, defines how big an attriubte-value list can get. This is the limit on how big an attribute-value list can be created using the `ATTR_LIST` or `attr_create_list()` macros.

```
Attr_avlist
attr_create_list(attrs)
```

The return value is used as the value for the attribute `ATTR_LIST`.

CANVAS_EACH_PAINT_WINDOW()

Macro providing built-in support for iteration across all the paint windows contained in a given canvas. Allows you to perform operations on multiple paint windows for which there are no canvas attributes.

CANVAS_END_EACH

Closes the loop started by `CANVAS_EACH_PAINT_WINDOW()`. These two macros are meant to be used together.

cursor_copy()

Creates and returns a copy of `src_cursor`.

```
Xv_Cursor
cursor_copy(src_cursor)
        Xv_Cursor  src_cursor;
```

Procedures and Macros

defaults_exists()

This function returns `TRUE` if the resource exists in the database via `XrmGetResource()`.

```
Bool
defaults_exists(name, class)
    char *name;
    char *class;
```

defaults_exists()

`defaults_set_boolean()` sets the resource to the value specified.

```
void
defaults_set_boolean(resource, value)
    char *resource;
    Bool  value;
```

defaults_get_boolean()

`defaults_get_boolean()` looks up the name-class pair in the resource database and returns `TRUE` if the value is one of:

- `True`
- `Yes`
- `On`
- `Enabled`
- `Set`
- `Activated`
- `1`

It returns `FALSE` if the value is one of:

- `False`
- `No`
- `Off`
- `Disabled`
- `Reset`
- `Cleared`
- `Deactivated`
- `0`

If the value is none of the previous, a warning message will be displayed and the default value will be returned. If the resource is not found, no error message is printed but the default value is still returned.

```
Bool
defaults_get_boolean(name, class, default_value)
    char *name, *class;
    int   default_value;
```

defaults_get_character()

`defaults_get_character()` looks up the name-class pair in the resource database and returns the resulting character value. If the resource is not found, then the default character value is returned.

```
char
defaults_get_character(name, class, default_char)
    char *name;
    char *class;
    char  default_char;
```

defaults_get_enum()

`defaults_get_enum()` looks up the value associated with name and class and scans the `pairs` table and returns the associated value. If no match is found, an error is generated and the value associated with the last entry is returned. `defaults_get_enum()` calls `defaults_get_string()` and determines the value returned by calling `defaults_lookup()`, passing the returned string as the `name` parameter.

```
int
defaults_get_enum(name, class, pairs)
    char           *name;
    char           *class;
    Defaults_pairs *pairs;
```

defaults_get_integer()

`defaults_get_integer()` looks up the name-class pair in the resource database and returns the resulting integer value. If the database does not contain the resource, the default value is returned.

```
int
defaults_get_integer(name, class, default_value)
    char *name;
    char *class;
    int   default_value;
```

defaults_get_integer_check()

`defaults_get_integer_check()` looks up the name-class pair in the resource database and returns the resulting integer value. If the value in the database is not between the values `minimum` and `maximum` (inclusive), an error message is printed and the default value is returned. If the resource is not found, no error message is printed but the default value is returned.

```
int
defaults_get_integer_check(name, class, default_value,
                           minimum, maximum)
    char *name;
    char *class;
    int   default_value;
    int   minimum;
    int   maximum;
```

Procedures and Macros

defaults_get_string()

`defaults_get_string()` returns the string value associated with the specified name-class pair in the resource database. If the resource is not found, the default string value is returned. `defaults_get_string()` returns a pointer into a static buffer maintained by the defaults package. Applications should not attempt to free this pointer. This buffer will be overwritten by the next call to the defaults package, so the application should maintain a copy if necessary.

```
char *
defaults_get_string(name, class, default_str)
    char *name;
    char *class;
    char *default_str;
```

defaults_init_db()

This function is called automatically by `xv_init()`, so it need not be called by your application. `defaults_init_db()` calls `XrmInitialize()`.

```
void
defaults_init_db()
```

defaults_load_db()

`defaults_load_db()` loads the database residing in the specified filename or the server database if filename is `NULL`. The database found in `filename` is loaded via `XrmGetFileDatabase()` and is merged into the existing resource database via `XrmMergeDatabases()`.

```
void
defaults_load_db(filename)
    char *filename;
```

defaults_lookup()

`defaults_lookup()` linearly scans the `pairs` array looking for `name`. The value associated with `name` is returned. The `pairs` array *must* contain a last element with a `NULL` `name` and a legal value associated with it. This value is returned if `name` does not match the `name` field of any of the elements in the `pairs` parameter.

```
int
defaults_lookup(name, pairs)
    char           *name;
    Defaults_pairs *pairs;
```

defaults_set_character()

`defaults_set_character()` sets the resource to the character value.

```
void
defaults_set_character(resource, character)
    char *resource;
    char  character;
```

defaults_set_integer()

`defaults_set_integer()` sets the resource to the value specified.

```
void
defaults_set_integer(resource, value)
    char *resource;
    int   value;
```

defaults_set_string()

`defaults_set_string()` sets the resource to the specified string.

```
void
defaults_set_string(resource, string)
    char *resource;
    char *string;
```

defaults_store_db()

```
void
defaults_store_db(filename)
    char *filename;
```

This function writes the database to the specified file via `XrmPutFileDatabase()`. This must be done in order to ensure that the database is accessible the next time the server is started.

dnd_decode_drop()

This function initiates a drag and drop data transfer using the selection mechanism. The drop event is decoded in three steps: First, the selection defined within the drop event is associated with the selection object passed into `dnd_decode_drop()`; second, the function sends an acknowledgement to the source, informing it that the transaction has been initiated; finally, if the drop-site is valid, the drop-site item that was dropped on is returned. Otherwise, if `dnd_decode_drop()` fails to initiate the drag and drop transaction, `XV_ERROR` is returned.

```
Xv_drop_site
dnd_decode_drop(sel_object, drop_event)
   Selection_requestor  sel_object;
   Event               *drop_event;
```

`sel_object` is an instantiated selection requestor object. `drop_event` is a pointer to the event structure that contains the trigger event. This event is either the semantic event `ACTION_DRAG_MOVE` or `ACTION_DRAG_COPY`, indicating the type of the drag and drop: a move or a copy.

Procedures and Macros

dnd_done()

This function is called by the application that is receiving the drop. It is called *after* the drop operation is completed. The function informs the toolkit that the drag and drop operation has been completed.

```
dnd_done(sel_object)
   Selection_requestor  sel_object;
```

`sel_object` is the `Selection_requestor` object that was previously passed into `dnd_decode_drop()`.

dnd_is_forwarded()

Allows applications to detect when a drop event has been forwarded from some other drop-site. Generally this happens when the user drops on the window manager's decor window or on an icon with the attribute `DROP_SITE_DEFAULT` set. The corresponding preview/drop event will have the flags field set with the `DND_FORWARDED` flag. This is tested for by using `dnd_is_forwarded(event)`.

In general, if the application handles previewing, it should check to see if the preview event was forwarded and not invert/highlight the drop site.

```
dnd_is_forwarded(event)
```

dnd_is_local()

Returns whether the `ACTION_DRAG_COPY` or `ACTION_DROP_MOVE` event was generated locally within the client.

```
dnd_is_local(event)
```

dnd_send_drop()

This initiates a drag. After the drag and drop object is created, an application calls `dnd_send_drop()`. This changes the cursor to the drag cursor, sends preview events to valid drop-sites and sends the trigger event (if a valid drop-site is dropped on). Trigger events are either `ACTION_DRAG_COPY` or `ACTION_DRAG_MOVE`.

A drag operation can be aborted by hitting the STOP key, or its equivalent. `dnd_send_drop()` reverts the drag cursor back to its normal state and returns `DND_ABORTED` when it detects the STOP key event. Application code needs to handle the `DND_ABORTED` after `dnd_send_drop()` returns. See *<xview/dragdrop.h>* for other valid return values.

```
int
dnd_send_drop(object)
    Dnd   object;
```

`object` is a `Dnd` object that describes the source data (an instance of `DRAGDROP`).

event_action()

Returns the semantic action from the event structure. For example, when the user selects the SELECT button, it returns `ACTION_SELECT`. If there is no action associated with an event, `event_action()` returns the value returned by `event_id()` for the event. The macro is passed a pointer to an `Event` structure.

event_alt_is_down()

Returns TRUE or FALSE, depending on the state of the Alt key. The macro is passed a pointer to an Event structure.

event_button_is_down()

Returns TRUE or FALSE, depending on the state of mouse buttons. Used in conjunction with: event_is_button(), event_left_is_down(), event_middle_is_down(), event_right_is_down(). The macro is passed a pointer to an Event structure.

event_ctrl_is_down()

Returns TRUE or FALSE, depending on the state of the Control key. Indicates the state of the Control key. The macro is passed a pointer to an Event structure.

event_id()

Returns the actual event ID, such as MS_LEFT to indicate the left mouse button. The macro is passed a pointer to an Event structure. Use of the macro event_action(), which returns the semantic action, is recommended instead of event_id().

event_is_ascii()

Returns TRUE or FALSE, depending whether or not the event is an ASCII key. The macro is passed a pointer to an Event structure.

event_is_button()

Returns TRUE or FALSE, depending whether or not the event is a button event. This procedure is used with: event_button_is_down(), event_left_is_down(), event_middle_is_down(), and event_right_is_down(). The macro is passed a pointer to an Event structure.

event_is_down()

Returns TRUE or FALSE, depending on the state of a mouse button or keyboard key. The macro is passed a pointer to an Event structure. Also see the macro event_is_up().

event_is_iso()

Returns TRUE or FALSE, depending whether a key event is within the ISO character set. The macro is passed a pointer to an Event structure.

event_is_key_bottom()

Returns TRUE or FALSE, depending whether a key event is within the left function keys. The macro is passed a pointer to an Event structure.

event_is_key_left()

Returns TRUE or FALSE, depending whether a key event is within the left function keys. This is for function keys on keyboards that are sectioned into four sets of fifteen function keys. The macro is passed a pointer to an Event structure.

event_is_key_right()

Returns TRUE or FALSE, depending whether a key event is within the right function keys. This is for function keys on keyboards that are sectioned into four sets of fifteen function keys. The macro is passed a pointer to an Event structure.

event_is_key_top()

Returns TRUE or FALSE, depending whether a key event is within the top function keys. This is for function keys on keyboards that are sectioned into four sets of fifteen function keys. The macro is passed a pointer to an Event structure.

event_is_meta()

Returns TRUE or FALSE, and determines whether event is a Meta key. The macro is passed a pointer to an Event structure.

event_is_string()

Returns TRUE or FALSE, determines if a string is associated with the event. The macro is passed a pointer to an Event structure.

event_is_up()

Returns TRUE or FALSE, depending on the state of the mouse button or keyboard in a particular event. The macro is passed a pointer to an Event structure. Also see the macro event_is_down().

event_left_is_down()

Returns TRUE or FALSE, and determines the state of the left mouse button. Used to determine the state of particular buttons for a three button mouse. This procedure is used in conjunction with: event_button_is_down(), event_is_button(), event_middle_is_down(), and event_right_is_down(). The macro is passed a pointer to an Event structure.

event_meta_is_down()

Indicates the state of the Meta key. The macro is passed a pointer to an Event structure.

event_middle_is_down()

Returns TRUE or FALSE, and determines the state of the middle mouse button. This procedure is used with: event_button_is_down(), event_is_button(), event_left_is_down(), and event_right_is_down(). The macro is passed a pointer to an Event structure.

event_right_is_down()

Returns TRUE or FALSE, and determines the state of the right mouse button. This procedure is used with: event_button_is_down(), event_is_button(), event_left_is_down(), and event_middle_is_down(). The macro is passed a pointer to an Event structure.

event_shift_is_down()

Indicates the state of the Shift key. The macro is passed a pointer to an Event structure.

event_string()

Used to determine the string value associated with the event ID. For normal ASCII event codes, `event_string()` returns a string value of the key. However, keys, ASCII or not, may be rebound to a string using `XRebindKeysym()`. The macro is passed a pointer to an `Event` structure.

event_time()

Returns the time field from the event structure. Used to determine the time value associated with the event. The macro is passed a pointer to an `Event` structure.

event_window()

Gets the window in which a particular event took place. The macro is passed a pointer to an `Event` structure.

event_xevent()

Returns a pointer to the X event structure associated with an XView event structure. The macro is passed a pointer to an `Event` structure.

event_xevent_type()

Returns the `type` field of the X event structure. The macro is passed a pointer to an `Event` structure.

frame_get_rect()

Gets the `rect` of the frame. `x,y` is the upper-left corner of the window coordinate space. Width and height attempt to include the window manager decoration.

```
void
frame_get_rect(frame, rect)
        Frame   frame;
        Rect   *rect;
```

frame_set_rect()

Sets the rect of the frame. `x,y` is the upper-left corner of the window coordinate space. Width and height attempt to include the window manager decoration.

```
void
frame_set_rect(frame, rect)
        Frame   frame;
        Rect   *rect;
```

MENUITEM_SPACE

Used to create a blank menu item.

```
menu_item = xv_create(NULL, MENUITEM_SPACE, NULL);
```

Procedures and Macros

menu_return_item()

Predefined notify procedure which, if given as the value for `MENU_NOTIFY_PROC`, causes the `menu_done_proc()`, if any, to return the handle of the selected item.

```
Menu_item
menu_return_item(menu, menu_item)
        Menu       menu;
        Menu_item  menu_item;
```

menu_return_value()

Predefined notify procedure which, if given as the value for `MENU_NOTIFY_PROC`, causes the `menu_done_proc()`, if any, to return the value of the selected item.

```
Xv_opaque
menu_return_value(menu, menu_item)
        Menu       menu;
        Menu_item  menu_item;
```

menu_show()

Displays the specified menu.

```
void
menu_show(menu, window, event, NULL)
        Menu        menu;
        Xv_Window   window;
        Event      *event;
```

`window` is the handle of the window over which the menu is displayed. `event` is the event which causes the menu to come up. The final `NULL` is required for private usage.

notice_prompt()

This is an XView Version 2 compatibility procedure. If you are creating a new notice for XView Version 3, use the `NOTICE` package. Refer to Chapter 12, *Notices*, in the *XView Programming Manual* for details. To obtain more information on `notice_prompt()`, see Appendix B, *The notice_prompt Function*, in the *XView Programming Manual*.

The function displays the notice and does not return until the user pushes a button or until its trigger or the default action has been seen. It returns a value of `NOTICE_FAILED` if `notice_prompt()` fails for any reason; otherwise, it is equivalent to the ordinal value of the button which caused the return (i.e., button actually selected or default button if default action triggered return). The `client_window` should be the window for which the notice has been generated. This is important for fonts and positioning information. For explicit positioning of the notice, clients should specify `NOTICE_FOCUS_XY`. The event, if not `NULL`, will be completely filled in at the time the `notice_prompt()` returns.

The possible status values that may be returned from this function are:

- The `int value` passed with every `NOTICE_BUTTON` attribute.
- `NOTICE_YES` if a button created with `NOTICE_BUTTON_YES` is pushed.
- `NOTICE_NO` if a button created with `NOTICE_BUTTON_NO` is pushed.

- `NOTICE_TRIGGERED` if a trigger was used.
- `NOTICE_FAILED` if the notice failed to pop up.

```
int
notice_prompt(window, event, attributes)
               XV_Window        window;
               Event           *event;
               attribute-list    attributes;
```

notify_default_wait3()

Predefined function you can register with the Notifier via the `notify_set_wait3_func()` call. Causes the required housekeeping to be performed on the process identified by `pid` when it dies. See the `wait`(2) man page for details of the `wait` and `rusage` structures.

```
Notify_value
notify_default_wait3(client, pid, status, rusage)
    Notify_client  client;
    int            pid;
    int           *status;
    struct rusage *rusage;
```

notify_dispatch()

Provided to allow programs which are not notification-based to run in an event-driven environment. Called regularly from within the application's main loop to allow the Notifier to go once around its internal loop and dispatch any pending events.

```
Notify_error
notify_dispatch()
```

notify_do_dispatch()

Enables *implicit dispatching*, in which the Notifier dispatches events from within calls to `read`(2) or `select`(2).

```
Notify_error
notify_do_dispatch()
```

notify_enable_rpc_svc()

If you use an RPC server that also needs to work with XView, use the XView function `notify_enable_rpc_svc()`, and do not call `svc_run()`. This function takes an **int** that tells the notifier whether it should handle RPC requests.

Using this aproach, `xv_main_loop()` handles incoming RPC requests; dispatching them just as if `svc_run()` had been called.

If `notify_enable_rpc_svc` is enabled, the Notifier checks `svc_fdset` and calls `svc_getreqset()` if a descriptor is readable (a request is coming in). Performance is affected by the additional call to `svc_getreqset()`.

```
void
notify_enable_rpc_svc(bool)
    int bool;
```

Procedures and Macros

notify_flush_pending()

Notifier removes client and flushes requests for it.

```
void
notify_flush_pending(nclient)
    Notify_client  nclient;
```

See Also: `notify_no_dispatch()`.

notify_get_destroy_func()

```
Notify_func
notify_get_destroy_func(client)
       Notify_client     client;
```

notify_get_event_func()

```
Notify_func
notify_get_event_func(client, when)
       Notify_client     client;
       int               when;
```

notify_get_exception_func()

```
Notify_func
notify_get_exception_func(client, fd)
       Notify_client     client;
       int               fd;
```

notify_get_input_func()

```
Notify_func
notify_get_input_func(client, fd)
       Notify_client     client;
       int               fd;
```

notify_get_itimer_func()

```
Notify_func
notify_get_itimer_func(client, which)
       Notify_client       client;
       int                 which;
```

notify_get_output_func()

```
Notify_func
notify_get_output_func(client, fd)
       Notify_client     client;
       int               fd;
```

notify_get_signal_func()

```
Notify_func
notify_get_signal_func(client, signal, mode)
       Notify_client      client;
       int                signal;
       Notify_signal_mode mode;
```

notify_get_wait3_func()

```
Notify_func
notify_get_wait3_func(client, pid)
       Notify_client     client;
       int               pid;
```

notify_interpose_destroy_func()

Interposes `destroy_func()` in front of `client`'s destroy event handler.

```
Notify_error
notify_interpose_destroy_func(client, destroy_func)
    Notify_client  client;
    Notify_func    destroy_func;
```

`client` is the handle of the Notifier client in front of which you are interposing. `destroy_func` is a notify function to be called before the client's destroy function. The format for the destroy function is:

```
Notify_value
destroy_func(client, status)
    Notify_client   client;
    Destroy_status  status;
```

notify_interpose_event_func()

Interposes `event_func()` in front of `client`'s event handler.

```
Notify_error
notify_interpose_event_func(client, event_func, type)
    Notify_client      client;
    Notify_func        event_func;
    Notify_event_type  type;
```

`client` is the handle of the Notifier client in front of which you are interposing. `event_func` is the notify function to be called before the client's event function. `type` is either `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`. The format for the event function is:

```
Notify_value
event_func(client, event, arg, type)
    Notify_client      client;
    Notify_event       event;
    Notify_arg         arg;
    Notify_event_type  type;
```

notify_interpose_exception_func()

Interposes `exception_func` in front of the client's exception handler.

```
Notify_error
notify_interpose_exception_func(client, exception_func,fd)
    Notify_client  client;
    Notify_func    exception_func;
    int            fd;
```

`client` is the handle of the Notifier client in front of which you are interposing. `exception_func` is the notify function to be called before the client's exception function. `type` is either `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`. The format for the exception function is:

```
Notify_value
exception_func(client, fd)
    Notify_client  client;
    int            fd;
```

notify_interpose_input_func()

Interposes `input_func` in front of `client`'s Notifier.

```
Notify_error
notify_interpose_input_func(client, input_func, fd)
    Notify_client  client;
    Notify_func    input_func;
    int            fd;
```

`client` is the handle of the Notifier client in front of which you are interposing. `input_func` is the notify function to be called before the client's input function. The format for the input function is:

```
Notify_value
input_func(client, fd)
    Notify_client  client;
    int            fd;
```

notify_interpose_itimer_func()

Interposes the `itimer_func` in front of the `client`'s timeout event handler.

```
Notify_error
notify_interpose_itimer_func(client, itimer_func, which)
    Notify_client  client;
    Notify_func    itimer_func;
    int            which;
```

`client` is the handle of the Notifier client in front of which you are interposing. `itimer_func` is the notify function to be called before the client's timeout handler. `which` is either `ITIMER_REAL` or `ITIMER_VIRTUAL`. The format for the itimer function is:

```
Notify_value
itimer_func(client, which)
    Notify_client  client;
    int            which;
```

notify_interpose_output_func()

Interposes `output_func` in front of `client`'s Notifier.

```
Notify_error
notify_interpose_output_func(client, output_func, fd)
    Notify_client  client;
    Notify_func    output_func;
    int            fd;
```

`client` is the handle of the Notifier client in front of which you are interposing. `output_func` is the notify function to be called before the client's output function. The format for the output function is:

```
Notify_value
output_func(client, fd)
    Notify_client  client;
    int            fd;
```

notify_interpose_signal_func()

Interposes the `signal_func()` in front of the signal event handler.

```
Notify_error
notify_interpose_signal_func(client, signal_func, signal, mode)
    Notify_client   client;
    Notify_func     signal_func;
    int             signal;
    int             mode;
```

`client` is the handle of the Notifier client in front of which you are interposing. `signal_func` is the notify function to be called before the client's signal handler. `signal` is the UNIX software interrupt. `mode` is either `NOTIFY_ASYNC` or `NOTIFY_SYNC`. The format for the signal function is:

```
Notify_value
signal_func(client, signal, mode)
    Notify_client   client;
    int             signal;
    int             mode;
```

notify_interpose_wait3_func()

Interposes the `wait3_func` in front of the Notifier.

```
Notify_error
notify_interpose_wait3_func(client, wait3_func, pid)
    Notify_client  client;
    Notify_func    wait3_func;
    int            pid;
```

`client` is the handle of the Notifier client in front of which you are interposing. `wait3_func` is the notify function to be called before the client's wait3 function. The format for the wait3 function on BSD-based systems is:

```
Notify_value
wait3_func(client, pid, status, rusage)
    Notify_client      client;
    int                pid;
```

```
    union wait        *status;
    struct rusage     *rusage;
```

The format for the wait3 function on SYSV-based systems is:

```
Notify_value
wait3_func(client, pid, status, rusage)
    Notify_client   client;
    int             pid;
    int            *status;
    struct rusage  *rusage;
```

notify_itimer_value()

Returns the current state of an interval timer for `client` in the structure pointed to by `value`. The `which` parameter is either `ITIMER_REAL` or `ITIMER_VIRTUAL`.

```
Notify_error
notify_itimer_value(client, which, value)
    Notify_client     client;
    int               which;
    struct itimerval *value;
```

notify_next_destroy_func()

Calls the next destroy event handler for `client`.

```
Notify_value
notify_next_destroy_func(client, status)
    Notify_client    client;
    Destroy_status   status;
```

`status` is one of:

```
DESTROY_PROCESS_DEATH
DESTROY_CHECKING
DESTROY_CLEANUP
DESTROY_SAVE_YOURSELF
```

notify_next_event_func()

Calls the next event handler for `client`.

```
Notify_value
notify_next_event_func(client, event, arg, type)
    Notify_client      client;
    Notify_event      *event;
    Notify_arg         arg;
    Notify_event_type  type;
```

`type` is either `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`.

notify_next_exception_func()

```
Notify_value
notify_next_exception_func(client, fd)
       Notify_client     client;
       int               fd;
```

notify_next_input_func()

```
Notify_value
notify_next_input_func(client, fd)
       Notify_client     client;
       int               fd;
```

notify_next_itimier_func()

```
Notify_value
notify_next_itimer_func(client, which)
       Notify_client       client;
       int                 which;
```

notify_next_output_func()

```
Notify_value
notify_next_output_func(client, fd)
       Notify_client     client;
       int               fd;
```

notify_next_signal_func()

```
Notify_value
notify_next_signal_func(client, signal, mode)
       Notify_client       client;
       int                 signal;
       Notify_signal_mode  mode;
```

notify_next_wait3_func()

BSD

```
Notify_value
notify_next_wait3_func(client, pid, status, rusage)
       Notify_client     client;
       int               pid;
       union wait       *status;
       struct rusage    *rusage;
```

notify_next_wait3_func()

SYSV based systems

```
Notify_value
notify_next_wait3_func(client, pid, status, rusage)
       Notify_client     client;
       int               pid;
       int              *status;
       struct rusage    *rusage;
```

notify_no_dispatch()

Prevents the Notifier from dispatching events from within the call to `read` (2) or `select` (2).

```
Notify_error
notify_no_dispatch()
```

See Also: `notify_do_dispatch()`.

notify_perror()

Analogous to the UNIX `perror` (3) system call. `s` is printed to `stderr`, followed by a terse description of `notify_errno()`.

```
void
notify_perror(s)
   char *s;
```

notify_post_event()

Posts a client event. A client event may be posted to the Notifier at any time. The poster of a client event may suggest to the Notifier when to deliver the event, but this is only a hint. The Notifier will see to it that it is delivered at an appropriate time (more on this below).

```
typedef char * Notify_event;

Notify_error
notify_post_event(client, event, when_hint)
    Notify_client      client;
    Notify_event       event;
    Notify_event_type  when_hint;
```

The `client` handle from `notify_set_event_func()` is passed to `notify_post_event()`. `event` is defined and interpreted solely by the `client`. `when_hint` is `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`. A return code of `NOTIFY_OK` indicates that the notification has been posted. Other values indicate an error condition. `NOTIFY_UNKNOWN_CLIENT` indicates that `client` is unknown to the Notifier. `NOTIFY_NO_CONDITION` indicates that `client` has no client event handler registered with the Notifier.

notify_post_event_and_arg()

Passes additional data with an event. For example, when the scrollbar posts an event to its owner to do a scroll, the scrollbar's handle is passed as an argument along with the event. The function `notify_post_event_and_arg()` provides the argument-passing mechanism.

```
Notify_error
notify_post_event_and_arg(client, event, when_hint, arg,
                          copy_func, release_func)
    Notify_client      client;
    Notify_event       event;
    Notify_event_type  when_hint;
    Notify_arg         arg;
    Notify_copy        copy_func;
    Notify_release     release_func;

typedef caddr_t Notify_arg;
typedef Notify_arg (*Notify_copy)();
#define NOTIFY_COPY_NULL   ((Notify_copy)0)

typedef void (*Notify_release)();
#define NOTIFY_RELEASE_NULL  ((Notify_release)0)
```

`copy_func()` is called to copy `arg` (and, optionally, `event`) when `event` and `arg` need to be queued for later delivery. `release_func()` is called to release the storage allocated during the copy call when `event` and `arg` were no longer needed by the Notifier.

Any of `arg`, `copy_func()`, or `release_func()` may be `NULL`. If `copy_func` is not `NOTIFY_COPY_NULL` and `arg` is `NULL`, then `copy_func()` is called anyway. This allows `event` the opportunity to be copied because `copy_func()` takes a pointer to `event`. The event pointed to may be replaced as a side effect of the copy call. The same applies to a `NOTIFY_RELEASE_NULL` release function with a `NULL` `arg` argument.

The `copy()` and `release()` routines are client-dependent, so you must write them yourself. Their calling sequences are the following:

```
Notify_arg
copy_func(client, arg, event_ptr)
    Notify_client  client;
    Notify_arg     arg;
    Notify_event  *event_ptr;

void
release_func(client, arg, event)
    Notify_client  client;
    Notify_arg     arg;
    Notify_event   event;
```

Procedures and Macros

notify_remove_destroy_func()

```
Notify_error
notify_remove_destroy_func(client, destroy_func)
       Notify_client     client;
       Notify_func       destroy_func;
```

notify_remove_event_func()

```
Notify_error
notify_remove_event_func(client, event_func, when)
       Notify_client     client;
       Notify_func       event_func;
       int               when;
```

notify_remove_exception_func()

```
Notify_error
notify_remove_exception_func(client, exception_func, fd)
       Notify_client     client;
       Notify_func       exception_func;
       int               fd;
```

notify_remove_input_func()

```
Notify_error
notify_remove_input_func(client, input_func, fd)
       Notify_client     client;
       Notify_func       input_func;
       int               fd;
```

notify_remove_itimer_func()

```
Notify_error
notify_remove_itimer_func(client, itimer_func, which)
       Notify_client       client;
       Notify_func         itimier_func;
       int                 which;
```

notify_remove_output_func()

```
Notify_error
notify_remove_output_func(client, output_func, fd)
       Notify_client     client;
       Notify_func       output_func;
       int               fd;
```

notify_remove_signal_func()

```
Notify_error
notify_remove_signal_func(client, signal_func, signal, mode)
       Notify_client       client;
       Notify_func         signal_func;
       int                 signal;
       Notify_signal_mode  mode;
```

notify_remove_wait3_func()

```
Notify_error
notify_remove_wait3_func(client, pid)
       Notify_client     client;
       Notify_func       wait3_func;
       int               pid;
```

notify_set_destroy_func()

Registers `destroy_func()` with the Notifier. `destroy_func()` is called when a destroy event is posted to `client` or when the process receives a `SIGTERM` signal.

```
Notify_func
notify_set_destroy_func(client, destroy_func)
    Notify_client  client;
    Notify_func    destroy_func;
```

The format for the destroy function is:

```
Notify_value
destroy_func(client, status)
    Notify_client   client;
    Destroy_status  status;
```

notify_set_event_func()

Registers the event handler `event_func()` with the Notifier.

```
Notify_error
notify_set_event_func(client, event_func, type)
    Notify_client      client;
    Notify_func        event_func;
    Notify_event_type  type;
```

`type` is either `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`.

The format for the event function is:

```
Notify_value
event_func(client, event, arg, type)
    Notify_client      client;
    Notify_event       event;
    Notify_arg         arg;
    Notify_event_type  type;
```

notify_set_exception_func()

Registers the exception handler `exception_func()` with the Notifier. The only known devices that generate exceptions at this time are stream-based socket connections when an out-of-band byte is available.

```
Notify_func
notify_set_exception_func(client, exception_func, fd)
    Notify_client  client;
    Notify_func    exception_func;
    int            fd;
```

The format for the exception function is:

```
Notify_value
exception_func(client, fd)
    Notify_client  client;
    int            fd;
```

notify_set_input_func()

Registers `input_func()` with the Notifier. `input_func()` will be called whenever there is input pending on `fd`.

```
Notify_func
notify_set_input_func(client, input_func, fd)
    Notify_client client;
    Notify_func   input_func;
    int           fd;
```

The format for the input function is:

```
Notify_value
exception_func(client, fd)
    Notify_client client;
    int           fd;
```

notify_set_itimer_func()

Registers the timeout event handler `itimer_func()` with the Notifier.

```
Notify_func
notify_set_itimer_func(client, itimer_func,
               which, value, ovalue)
    Notify_client      client;
    Notify_func        itimer_func;
    int                which;
    struct itimerval  *value, *ovalue;
```

The semantics of `which`, `value`, and `ovalue` parallel the arguments to `setitimer`(2).

`which` is either `ITIMER_REAL` or `ITIMER_VIRTUAL`.

The format for the itimer function is:

```
Notify_value
itimer_func(client, which)
    Notify_client  client;
    int            which;
```

notify_set_output_func()

Registers `output_func()` with the Notifier. `output_func()` will be called whenever output has been completed on `fd`.

```
Notify_func
notify_set_output_func(client, output_func, fd)
    Notify_client client;
    Notify_func   output_func;
    int           fd;
```

The format for the output function is:

```
Notify_value
exception_func(client, fd)
    Notify_client client;
    int           fd;
```

notify_set_signal_func()

Registers the signal event handler `signal_func()` with the Notifier. `signal_func()` will be called whenever `signal` is caught by the Notifier. `when` can be either `NOTIFY_SYNC` or `NOTIFY_ASYNC`.

Calling `notify_set_signal_func()` with a `NULL` in the place of the `signal_func()` turns off checking for that signal for that client.

```
Notify_func
notify_set_signal_func(client, signal_func, signal, when)
    Notify_client      client;
    Notify_func        signal_func;
    int                signal;
    Notify_signal_mode  when;
```

The format for the `signal_func` function is:

```
Notify_value
signal_func(client, sig, mode)
    Notify_client  client;
    int            sig;
    int            mode;
```

notify_set_wait3_func()

Registers the function `wait3_func()` with the Notifier. The registered function is called after `client`'s process identified by `pid` dies. To do the minimum processing, register the predefined function `notify_default_wait3()`.

```
Notify_func
notify_set_wait3_func(client, wait3_func, pid)
    Notify_client  client;
    Notify_func    wait3_func;
    int            pid;
```

The format for the wait3 function on BSD-based systems is:

```
Notify_value
wait3_func(client, pid, status, rusage)
    Notify_client   client;
    int             pid;
    union wait     *status;
    struct rusage  *rusage;
```

The format for the wait3 function on SYSV-based systems is:

```
Notify_value
wait3_func(client, pid, status, rusage)
    Notify_client  client;
    int            pid;
    int           *status;
    struct rusage *rusage;
```

notify_start()

Begins dispatching of events by the Notifier.

```
Notify_error
notify_start()
```

notify_stop()

Terminates dispatching of events by the Notifier.

```
Notify_error
notify_stop()
```

notify_veto_destroy()

Called from within a destroy event handler when `status` is `DESTROY_CHECKING` and the application does not want to be destroyed.

```
Notify_error
notify_veto_destroy(client)
    Notify_client client;
```

OPENWIN_EACH_VIEW()

Macro providing built-in support for iteration across all the views contained in a given openwin. Allows you to perform operations on multiple views for which there are no openwin attributes.

OPENWIN_END_EACH

Closes the loop started by the macro `OPENWIN_EACH_VIEW()`. These macros are meant to be used together, as in the following example:

```
OPENWIN_EACH_VIEW(openwin, view)
        Openwin         openwin;
        Openwin_item    item;
        xv_set(openwin, attributes, 0);
OPENWIN_END_EACH;
```

panel_advance_caret()

Advances the input focus to the next item that can accept input focus. If on the last input focus, rotate back to the first. Returns the new caret item or `NULL` if there are no input focus items.

```
Panel_item
panel_advance_caret(panel)
        Panel     panel;
```

panel_backup_caret()

Backs the caret up to the previous input focus item. If already on the first input focus item, rotate back to the last. Returns the new caret item or `NULL` if there are no input focus items.

```
Panel_item
panel_backup_caret(panel)
        Panel    panel;
```

PANEL_CHECK_BOX

Macro for "`PANEL_TOGGLE`, `PANEL_FEEDBACK`, `PANEL_MARK`." Creates non-exclusive choice item(s) with `check_boxes` instead of boxes.

```
xv_create(panel, PANEL_CHECK_BOX, NULL);
```

PANEL_CHECK_BOX expands to:

```
PANEL_TOGGLE,
PANEL_FEEDBACK,    PANEL_MARK
```

To use an ATTR_LIST argument, the ATTR_LIST must be the first attribute in an attribute-value list. See PANEL_CHOICE_STACK for an example.

PANEL_CHOICE_STACK

Macro for "PANEL_CHOICE, PANEL_DISPLAY_LEVEL, PANEL_CURRENT." Creates an OPEN LOOK abbreviated choice menu button.

```
xv_create(panel, PANEL_CHOICE_STACK, NULL);
```

PANEL_CHOICE_STACK expands to:

```
PANEL_CHOICE,
PANEL_DISPLAY_LEVEL,    PANEL_CURRENT
```

To use an ATTR_LIST argument, the ATTR_LIST must be the first attribute in an attribute-value list. You need to include an explicit PANEL_CHOICE, rather than the PANEL_CHOICE_STACK attribute. For example,

```
xv_create(owner, PANEL_CHOICE,
     ATTR_LIST, at,
      PANEL_DISPLAY_LEVEL,PANEL_CURRENT,
     .....,
      NULL);
```

PANEL_EACH_ITEM()

Macro to iterate over each item in a panel. The corresponding macro PANEL_END_EACH closes the loop opened by PANEL_EACH_ITEM().

```
PANEL_EACH_ITEM(panel, item)
       Panel       panel;
       Panel_item  item;
```

PANEL_END_EACH

Closes the loop started by PANEL_EACH_ITEM(). Same usage as OPENWIN_EACH_VIEW().

panel_paint()

Paints an item or an entire panel. paint_behavior can be either PANEL_CLEAR, which causes the area occupied by the panel or item to be cleared prior to painting, or PANEL_NO_CLEAR.

```
int
panel_paint(panel_object, paint_behavior)
       Xv_object       panel_object;
       Panel_setting   paint_behavior;
```

Note that panel_object may be a PANEL or a PANEL_ITEM. If panel_object does not exist, or panel_paint() is called with an invalid paint_behavior value, the function returns XV_ERROR; otherwise, it returns XV_OK.

Procedures and Macros

panel_text_notify()

Default notify procedure for panel text items. Determines what the correct Panel_setting value should be based on the event passed into the notify procedure. The Panel_setting value causes the PANEL_TEXT package to adjust the caret to advance on Return or Tab, caret to back up on Shift-Return or Shift-Tab, printable characters to be inserted into item's value, and all other characters to be discarded.

```
Panel_setting
panel_text_notify(item, event)
       Panel_item   item
       Event       *event
```

Returns PANEL_NEXT, PANEL_PREVIOUS, PANEL_INSERT, or PANEL_NONE, respectively.

PANEL_TOGGLE

Macro for "PANEL_CHOICE, PANEL_CHOOSE_ONE, FALSE." Used to create non-exclusive choice item(s).

```
xv_create(panel, PANEL_TOGGLE, NULL);
```

PANEL_TOGGLE expands to:

```
PANEL_CHOICE,
PANEL_CHOOSE_ONE,    FALSE
```

To use an ATTR_LIST argument, the ATTR_LIST must be the first attribute in an attribute-value list. See PANEL_CHOICE_STACK for an example.

rect_below()

Returns TRUE if r2 lies below r1 AND the left edge of r2 isn't greater than the right edge of r1 AND the right edge isn't less than the left edge of r1.

```
int
rect_below( r1, r2 )
   Rect *r1, *r2;
```

rect_borderadjust()

Used to adjust the borders. This macro takes a rect pointer and an int. This macro is defined as follows:

```
#define rect_borderadjust(r,m) \
          { (r)->r_width+=(m)+(m);(r)->r_height+=(m)+(m);}
```

rect_bottom()

Takes a rect pointer and returns the position of the bottom of the rect. This macro is defined as follows:

```
#define   rect_bottom(rect)   ((rect)->r_top+(rect)->r_height-1)
```

rect_bounding()

Return the rectangle that defines the region bound by rectangle `r1` and `r2`.

```
struct rect
rect_bounding( r1, r2 )
   struct rect *r1, *r2;
```

rect_clipvector()

Clip vector defined by coordinates `x1y1,x2y2` to space within rectangle `r`; returns `TRUE` if original vector resides within rectangle.

```
unsigned
rect_clipvector( r, x1, y1, x2, y2 )
struct rect *r;
   int x1, y1, x2, y2;
```

rect_construct()

Constructs a `rect` based on the values specified. It takes a `rect` pointer and four `int` arguments. This macro is defined as follows:

```
#define rect_construct(r,x,y,w,h) \
  {(r)->r_left=(x);(r)->r_top=(y);(r)->r_width=(w);(r)->r_height=(h);}
```

rect_distance()

Compute the distance from coordinate `xy` to rectangle. If `xy` is inside rectangle, `0` is returned. If `x_used` or `y_used` are non-zero, then the projection point is returned.

```
int
rect_distance( rect, x, y, x_used, y_used )
   Rect *rect;
   int  x, y;
   int  x_used, y_used;
```

rect_equal()

Compares two `rect` pointers and returns `TRUE` if all dimensions are equal. This macro is defined as follows:

```
#define rect_equal(r1,r2) \
    ((r1)->r_left==(r2)->r_left && (r1)->r_width==(r2)->r_width && \
     (r1)->r_top==(r2)->r_top && (r1)->r_height==(r2)->r_height)
```

rect_includespoint()

Returns `TRUE` if specified coordinates are within specified `rect`. It takes a `rect` pointer and two integers (x and y coordinates). This macro is defined as follows:

```
#define rect_includespoint(r,x,y) \
    ((x) >= (r)->r_left && (y) >= (r)->r_top && \
     (x)<(r)->r_left+(r)->r_width && (y)<(r)->r_top+(r)->r_height)
```

rect_includesrect()

Determines whether or not a specified `rect` is contained in another. It takes two `rect` pointers. This macro is defined as follows:

```
#define rect_includesrect(r1, r2) \
    ((r1)->r_left <= (r2)->r_left && (r1)->r_top <= (r2)->r_top && \
     (r1)->r_left+(r1)->r_width >= (r2)->r_left+(r2)->r_width && \
     (r1)->r_top+(r1)->r_height >= (r2)->r_top+(r2)->r_height)
```

rect_intersection()

Calculates the intersection of rectangles `r1` and `r2` and returns the resulting rectangle `r`.

```
struct rect
rect_intersection( r1, r2, r )
   Rect *r1, *r2, *r;
```

rect_intersectsrect()

Determines whether or not one `rect` intersects another. It takes two `rect` pointers. This macro is defined as follows:

```
#define rect_intersectsrect(r1,r2) \
    ((r1)->r_left<(r2)->r_left+(r2)->r_width && \
     (r1)->r_top<(r2)->r_top+(r2)->r_height &&  \
     (r2)->r_left<(r1)->r_left+(r1)->r_width && \
     (r2)->r_top<(r1)->r_top+(r1)->r_height)
```

rect_isnull()

Takes a `rect` pointer and returns `TRUE` if either the width or the height of the rect is 0. Otherwise returns `FALSE`. This macro is defined as follows:

```
#define rect_isnull(r)     ((r)->r_width == 0 || (r)->r_height == 0
```

rect_marginadjust()

Adjusts the margins in a `rect`. It takes a `rect` pointer and an integer. This macro is defined as follows:

```
#define rect_marginadjust(r,m)                              \
          { (r)->r_left-=(m);(r)->r_top-=(m);               \
           (r)->r_width+=(m)+(m);(r)->r_height+=(m)+(m);}
```

rect_order()

Return TRUE if rectangles r1 and r2 are in specified sort order. Where sort order is defined by:

```
RECTS_TOPTOBOTTOM: returns TRUE if r1 top <= r2 top
RECTS_BOTTOMTOTOP: returns TRUE if r1 bottom >= r2 bottom
RECTS_LEFTORIGHT:  returns TRUE if r1 left <= r2 left
RECTS_RIGHTTOLEFT: returns TRUE if r1 right >= r2 right
RECTS_UNSORT:      returns TRUE
```

```
unsigned
rect_order( r1, r2, sortorder )
   struct rect *r1, *r2;
   int   sortorder;
```

rect_passtochild()

Takes two integers (x and y coordinates) and a rect pointer. This macro is defined as follows:

```
#define   rect_passtochild(x,y,rect) \
  {(rect)->r_left=(rect)->r_left-(x); (rect)->r_top=(rect)->r_top-(y);}
```

rect_passtoparent()

Takes two integers (x and y coordinates) and a rect pointer. This macro is defined as follows:

```
#define   rect_passtoparent(x,y,rect) \
  {(rect)->r_left=(rect)->r_left+(x); (rect)->r_top=(rect)->r_top+(y);}
```

rect_print()

Takes a rect pointer and prints the rectangle on stderr.

```
#define rect_print(rect)
\
  (void)fprintf(stderr,"[left: %d, top: %d, width: %d, height: %d]0,
\
  (rect)->r_left, (rect)->r_top, (rect)->r_width, (rect)->r_height)
```

rect_right()

Takes a rect pointer and returns the position of the right edge of the rect. This macro is defined as follows:

```
#define   rect_right(rect)    ((rect)->r_left+(rect)->r_width-1)
```

rect_right_of()

Returns TRUE if r2 lies to the right of r1 AND the bottom of r2 isn't above the top of r1 AND the top of r2 isn't below the bottom of r1.

```
int
rect_right_of( r1, r2 )
   Rect *r1, *r2;
```

Procedures and Macros

rect_sizes_differ()

Takes two `rect` pointers. If all dimensions are equal, it returns `FALSE`, if not it returns `TRUE`. This macro is defined as follows:

```
#define rect_sizes_differ(r1, r2) \
   ((r1)->r_width != (r2)->r_width || (r1)->r_height != (r2)->r_height)
```

SCROLLABLE_PANEL

Used to create a scrollbar panel. To add a scrollbar after creating the panel, create a scrollbar with the panel as its parent. For example:

```
panel = xv_create(frame, SCROLLABLE_PANEL, NULL);
xv_create(panel, SCROLLBAR, NULL);
```

Note: Scrollable panels are not inherently OPEN LOOK-compliant.

scrollbar_paint()

Repaints all portions of the scrollbar.

```
void
scrollbar_paint(scrollbar)
     Scrollbar   scrollbar;
```

selection_*

Appendix B, *Selection Compatibility Procedures and Macros*, describes the `selection_*` procedures. The `selection_` procedures provide compatibility for applications using selections created prior to XView Version 3. If you are creating a new application, refer to Chapter 18, *Selections*, in the *XView Programming Manual*.

sel_convert_proc()

Default selection convert procedure provided by XView Version 3 for use with the `SELECTION_OWNER` package. This procedure allows the selection owner to communicate with the selection requestor.

```
int
sel_convert_proc(sel_owner, type, data, length, format)
   Selection_owner  sel_owner;
   Atom            *type;      /* Input/Output */
   Xv_opaque       *data;
   unsigned long   *length;    /* Output */
   int             *format;
```

`sel_owner` is the selection owner. `type` is the form data should be converted to. `data` is a pointer to the reply buffer. `length` is a pointer to the length of the reply buffer.

`format` is a pointer to an integer representing the number of bits in a single data member in reply buffer. Returns `TRUE` if completed successfully, `FALSE` if it does not complete successfully.

sel_post_req()

This procedure is used to send a non-blocking request to the selection owner. The application's reply procedure is called by the SELECTION package when the selection owner has sent its reply.

```
int
sel_post_req( sel )
   Selection_requestor  sel;
```

sel is the selection requestor. Returns XV_OK if it successfully sends the request, otherwise it returns XV_ERROR. When no reply procedure is defined, XV_ERROR is returned (see the description for the attribute SEL_REPLY_PROC).

textsw_add_mark()

Adds a new mark at position. flags can be either TEXTSW_MARK_DEFAULTS or TEXTSW_MARK_MOVE_AT_INSERT.

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

textsw_append_file_name()

Returns 0 if textsw is editing a file and, if so, appends the name of the file at the end of name.

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

textsw_delete()

Returns 0 if the operation fails. Removes the span of characters beginning with first and ending one before last_plus_one.

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

textsw_edit()

Returns 0 if the operation fails. Erases a character, a word or a line, depending on whether unit is SELN_LEVEL_FIRST or SELN_LEVEL_LINE. If direction is 0, characters after the insertion point are affected; otherwise, characters before the insertion point are affected. The operation will be done count times.

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

Procedures and Macros

textsw_erase()

Returns 0 if the operation fails. Equivalent to textsw_delete() but does not affect the global shelf.

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

textsw_file_lines_visible()

Fills in top and bottom with the file line indices of the first and last file lines being displayed in textsw.

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

textsw_find_bytes()

Beginning at the position addressed by first, searches for the pattern specified by buf of length buf_len. Searches forward if flags is 0, else searches backward. Returns -1 if no match, else matching span placed in indices addressed by first and last_plus_one.

```
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, flags;
```

textsw_find_mark()

Returns the current position of mark. If this operation fails, it will return TEXTSW_INFINITY.

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

textsw_first()

Returns the first textsw view.

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

textsw_index_for_file_line()

Returns the character index for the first character in the line given by `line`. If this operation fails, it will return `TEXTSW_CANNOT_SET`.

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

textsw_insert()

Inserts characters in `buf` into `textsw` at the current insertion point. The number of characters actually inserted is returned. This will equal `buf_len` unless there was a memory allocation failure. If there was a failure, it returns `NULL`.

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

textsw_match_bytes()

Searches for a block of text in the `textsw`'s contents; ends with characters matching `end_sym`. This function places the starting index of the matching block in `first` and its ending index in `last`.

```
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;
```

textsw_next()

Returns the next view in the set of `textsw` views.

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

textsw_normalize_view()

Repositions the text so that the character at `position` is visible and at the top of the subwindow.

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

textsw_possibly_normalize()

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

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

textsw_remove_mark()

Removes an existing mark from `textsw`.

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

textsw_replace_bytes()

Replaces the character span from `first` to `last_plus_one` with the characters in `buf`. The return value is the number of bytes inserted or deleted. The number is positive if bytes are inserted, negative if bytes are deleted. (The number is also negative if the original string is longer than the one that replaces it.) If this operation fails, it will return a value of `NULL`.

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

textsw_reset()

Discards edits performed on the contents of `textsw`. If needed, a message box will be displayed at `x,y`.

```
void
textsw_reset(textsw, x, y)
        Textsw   textsw;
        int      x, y;
```

textsw_save()

Saves any edits made to the file currently loaded into `textsw`. If needed, a message box will be displayed at `x,y`.

```
unsigned
textsw_save(textsw, x, y)
        Textsw   textsw;
        int      x, y;
```

textsw_screen_line_count()

Returns the number of screen lines in `textsw`.

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

textsw_scroll_lines()

Moves the text up or down by `count` lines. If `count` is positive, then the text is scrolled up on the screen. If negative, the text is scrolled down (backward in the file).

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

textsw_set_selection()

Sets the selection to begin at `first` and includes all characters up to `last_plus_one`. A `type` value of `1` indicates primary selection, `2` indicates secondary selection.

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

textsw_store_file()

Stores the contents of `textsw` to the file named by `filename`. If needed, a message box will be displayed at `x,y`.

```
unsigned
textsw_store_file(textsw, filename, x, y)
        Textsw     textsw;
        char      *filename;
        int        x, y;
```

ttysw_input()

Appends `len` number of characters from `buf` onto `tty`'s input queue. It returns the number of characters accepted.

```
int
ttysw_input(tty, buf, len)
        Tty   tty;
        char  *buf;
        int    len;
```

Procedures and Macros

ttysw_output()

Appends `len` number of characters from `buf` onto `tty`'s output queue; it sends them through the terminal emulator to the TTY. It returns the number of characters accepted.

```
int
ttysw_output(tty, buf, len)
        Tty    tty;
        char  *buf;
        int    len;
```

window_done()

Destroys the entire hierarchy to which `win` belongs.

```
int
window_done(win)
        Xv_Window  win;
```

window_fit()

This macro causes `win` to fit its contents in the dimensions specified with `WIN_FIT_HEIGHT` and `WIN_FIT_WIDTH`. It is defined as:

```
xv_set(win, WIN_FIT_HEIGHT, 0, WIN_FIT_WIDTH, 0, NULL)
```

window_fit_height()

Causes `win` to fit its contents in the height specified with `WIN_FIT_HEIGHT`. The macro is defined as:

```
xv_set(win, WIN_FIT_HEIGHT, 0, NULL)
```

window_fit_width()

Causes `win` to fit its contents in the width specified with `WIN_FIT_WIDTH`. The macro is defined as:

```
xv_set(win, WIN_FIT_WIDTH, 0, NULL)
```

window_read_event()

Reads the next input event for `window`. In case of error, sets the global variable `errno` and returns `-1`.

```
int
window_read_event(window, event)
        Xv_window  window;
        Event     *event;
```

win_set_kbd_focus

Assigns input focus (via `XSetInputFocus()`) to the passed window.

```
int
win_set_kbd_focus(window, xid)
    Xv_Window  window;  /* XView window to set focus on */
    XID        xid;     /* XID of XView window */
```

wmgr_bottom()

This procedure is a SunView Compatibility procedure. Sets stackmode to `Below` in `XConfigureWindow`.

```
void
wmgr_bottom(frame)
        Frame   frame;
```

wmgr_changelevel()

This procedure is a SunView Compatibility procedure. It sets the stackmode in `XConfigureWindow`.

```
void
wmgr_changelevel(window,parent,top)
        Xv_Window  window;
        int        parent;
        int        top;
```

wmgr_close()

Sets `wmhints.initial_state` to `ICONICSTATE`.

```
void
wmgr_close(frame)
        Frame  frame;
```

wmgr_completechangerect()

Calls `XConfigureWindow` with a new `rect`.

```
void
wmgr_completechangerect(window, rectnew, rectoriginal,
        parentprleft, parentprtop)
                Xv_window  window;
                Rect      *rectnew, *rectoriginal;
                int        parentprleft, parentprtop;
```

wmgr_open()

Sets `wmhints.initial_state` to `NORMALSTATE`.

```
void
wmgr_open(frame)
        Frame  frame;
```

wmgr_refreshwindow()

Calls `XConfigureWindow` repeatedly.

```
void
wmgr_refreshwindow(window)
        Xv_window  window;
```

Procedures and Macros

wmgr_top()

Sets stackmode to `Above` in `XConfigureWindow`.

```
void
wmgr_top(frame)
        Frame  frame;
```

xv_col()

Returns an integer representing the number of pixels, excluding the left margin of the window. This may be used in conjunction with a panel's `WIN_COL_GAG`.

```
int
xv_col(window, column)
        Xv_Window  window;
        int        column;
```

xv_cols()

Returns an integer representing the number of pixels, including the left margin of the window.

```
int
xv_cols(window, columns)
        Xv_Window  window;
        int        columns;
```

xv_create()

To create *any* XView object, call the generic procedure `xv_create`. This procedure will return a handle to some XView object. It takes as parameters the owner of the object being created, the type of object to create, and a list of attributes. The attribute list must terminate with a `NULL`. In specifying the type, you must use the name of some XView package. That name must be in all capital letters to distinguish it from the corresponding data type. The package is the name of the package to which the object you wish to create belongs.

The procedure `xv_create()` returns either the handle for the new object or `XV_NULL` if the attempt at object creation fails.

```
Xv_opaque
xv_create(owner, package, attributes)
        Xv_object        owner;
        Xv_pkg          *package;
        attribute-list   attributes;
```

xv_destroy()

To destroy an XView object and any subframes owned by that object, use the procedure `xv_destroy()`. It will return either `XV_ERROR` or `XV_OK`.

```
int
xv_destroy(object)
        Xv_object  object;
```

XV_DISPLAY_FROM_WINDOW()

Macro to get at the handle of the display structure from a window object.

```
Xv_opaque
XV_DISPLAY_FROM_WINDOW(window)
        Xv_Window  window;
```

xv_error()

Is called by the XView internals in the event of an error.

```
char *
xv_error(object, attributes)
    Xv_opaque        object;
    attribute-list   attributes;
```

xv_error_format()

This function calls a pointer to a static `char *` describing the XView error that has occurred. It should be copied into your buffer if you wish to retain the value since subsequent calls overwrite the contents.

```
char *
xv_error_format(object, avlist)
    Xv_object        object;
    Attr_avlist      avlist;
```

xv_find()

To find any XView object, use the procedure `xv_find()`. If the object is not found, `xv_find` will automatically attempt to create it.

```
Xv_opaque
xv_find(owner, package, attributes)
        Xv_object            owner;
        Xv_pkg              *package;
        attribute-list        attributes;
```

xv_get()

To get the value of any single attribute of *any* XView object, use the procedure `xv_get()`.

```
Xv_opaque
xv_get(object, attribute)
        Xv_object           object;
        attribute-list       attributes;
```

The procedure returns `0` for failure. As a result, you cannot detect errors when retrieving the values of attributes which might return `0` as a valid value. Note that, although you can supply `XV_NULL` as the owner when you create a new object with `xv_create`, you *must* provide the object when asking for the value of an attribute.

For example, you cannot just ask for:

```
xv_get(XV_NULL, XV_FONT)
```

to determine the default font, but you can ask for the font associated with a particular screen, as in:

```
xv_get(Screen1, XV_FONT)
```

Some attributes require a screen or server to be supplied to `xv_get()`. For such attributes, you should ask any window you think should be on the same screen what to identify as its screen or server, using the attributes `XV_SCREEN` and `XV_SERVER`.

If you know your application only runs on a single screen, you can use the global values `xv_default_server` and `xv_default_screen` or you can ask the defaults database what the server is.

xv_init()

This procedure performs many tasks including: opening the connection to the server, initializing the Notifier, and initializing the Resource Manager database (see Chapter 17, *Resources*, in the *XView Programming Manual*). By default `xv_init()` opens a connection to the server described by the `DISPLAY` environment variable. The server connections may be changed using appropriate command-line attribute-value pairs. No matter which server is used, `xv_init()` returns a handle to that server object. For further information concerning use with servers, see Chapter 15, *Nonvisual Objects*, in the *XView Programming Manual*.

```
int
main(argc, argv)
        int     argc;
        char  **argv:
{
/* initialization/declarations */
           .
           .
           .
  (void)xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, 0);
  frame = xv_create(NULL, FRAME, FRAME_LABEL, "foo", 0);
}
```

xv_row()

Returns an integer representing the number of pixels, excluding the top margin of the window. Used in conjunction with a panel's `WIN_ROW_GAP`.

```
int
xv_row(window, row)
        Xv_Window  window
        int        row;
```

xv_rows()

Returns an integer representing the number of pixels, including the top margin of the window.

```
int
xv_rows(window, rows)
        Xv_Window  window
        int        rows;
```

XV_SCREEN_FROM_WINDOW()

Macro to return the handle to the screen object from the window object.

```
Xv_screen
XV_SCREEN_FROM_WINDOW(window)
        Xv_Window  window;
```

xv_send_message()

Lets two separate processes communicate with each other. You can specify the `addressee` field either with one of the constants `XV_POINTER_WINDOW` or `XV_FOCUS_WINDOW` or with the window's `XID`, if known. If the `addressee` is `XV_POINTER`, then the message is sent to the window that the pointer is in. If the `addressee` is `XV_FOCUS_WINDOW`, then the message is sent to the window that currently has the focus, regardless of the pointer position. If the `addressee` is a window's `XID`, then the message is sent to the window with the corresponding ID.

```
void
xv_send_message(
        window, addressee, msg_type, format, data, len)
                Xv_object   window;
                Xv_opaque   addressee;
                char       *msg_type;
                int         format;
                Xv_opaque  *data;
                int         len;
```

`msg_type` is a pointer to a string specifying the property you want the Atom for. `format` is the number of bits of a single data member (8,16, or 32). `data` is a pointer to the data to send. `len` is the number of bytes of data to send.

XV_SERVER_FROM_WINDOW()

Macro returns the handle to the server object from a window object.

```
Xv_opaque
XV_SERVER_FROM_WINDOW(window)
        Xv_Window   window;
```

Procedures and Macros

xv_set()

To set the value of one or more attributes of *any* XView object, call the procedure `xv_set` with the handle to the object whose attributes you wish to set and a list of attribute-value pairs terminating in a `NULL`.

```
Xv_object
Xv_set(owner, type, attributes)
        Xv_object          owner;
        vu type            *type;
        attribute-list      attributes;
```

The procedure `xv_set()` returns `XV_OK` if it succeeds; otherwise, it returns an error code indicating that the attribute on which it ran encountered problems.

xv_super_set_avlist()

This function handles the parsing of attributes that are generic to a package's super set.

```
Xv_public    Xv_opaque
xv_super_set_avlist(object, pkg, avlist)
    register Xv_opaque   object;
    register Xv_pkg     *pkg;
    Attr_avlist          avlist;
```

xv_unique_key()

Generates a key for use with `XV_KEY_DATA`.

Warning: The uniqueness of any particular key is *not* guaranteed since the value returned is generated simply by incrementing a pre-defined "unique" value.

```
int
xv_unique_key()
```

xv_usage()

Prints a list of toolkit's generic command-line arguments.

```
void
xv_usage(app_name)
   char *app_name;
```

See Also: `XV_USAGE_PROC`

xv_window_loop()

The function `xv_window_loop()` maps the frame passed as an argument and makes other frames and windows in the application "deaf." This is similar to `FRAME_BUSY` except that the cursor does not change to a stopwatch and the frame header does not show the gray "busy" pattern. `xv_window_loop()` does not lock the screen.

```
Xv_opaque
xv_window_loop(frame)
    Frame      frame;
```

The frame passed into `xv_window_loop()` can have more than one subwindow of any type (this was a restriction with SunView's `window_loop()` function).

xv_window_loop() does not return until a call to xv_window_return().

```
void
xv_window_return(return_val)
    Xv_opaque  return_val;
```

This presumably occurs in a callback that originates from a button (for example) on the frame. The return value passed into xv_window_return() is the value returned by xv_window_loop(). Since the screen is *not* locked when xv_window_loop() is active, the user might be able to dismiss the frame using the window manager. Doing this causes the application to hang since xv_window_return() is not called. To avoid this behavior, you need to attach a destroy procedure to the frame. The destroy procedure should call xv_window_return().

xv_window_return()

See the procedure xv_window_loop().