XI PROGRAMMER'S REFERENCE

Published by:

Providence Software Solutions, Inc.
202 New Edition Court
Cary NC, 27511
919-854-1800
919-4393034 (Fax)
http://www.xvt.com (Website)

Table of Contents

Chapter 1

XI Data Structures and Types

This chapter details the contents of XI data structures and the values for defined types and constants. Most constants are defined using `enum`. However, when a constant must have a specific value, you will find it in a section with an asterisk(*) after the "root" part of the name. For example, "XI_ATR_*" refers to a set of defined values that all begin with "XI_ATR_".

COMBO_ICON — Icon Resource ID for Cell and Edit Field buttons

Summary

```
#ifndef COMBO_ICON
#define COMBO_ICON 1026
#endif
```

Description

This defines the resource ID for the icon that will be used for cell and edit field buttons. In general, this is a "down-arrow" icon. If you are going to use any of these options on the list, the appropriate icon must appear in your URL file. The URL for the example programs demonstrate how to do this.

This is the default value for the icon. It can be set to something else using **xi_set_pref**.

See Also

XI_PREF_COMBO_ICON, xi_set_pref

HSCROLL_CID_CONST Horizontal Scrollbar Control ID adjustment

Summary

```
#define HSCROLL_CID_CONST 6000
```

Description

If a list can scroll vertically, it creates an XVT scrollbar with the same control ID as the list. There is no conflict because the control ID's for XVT and XI controls are separate. However, if the list is also horizontally scrolling, an XVT scrollbar must be created for horizontal scrolling. This scrollbar will have a control ID of **HSCROLL_CID_CONST** plus the list ID.

If you are using other XVT controls in a window with XI controls, be sure that you do not use any of the same control ID's as are being used for horizontal and vertical scrollbars on an XI list.

XI_ATR_* Attributes

Summary

```
#define XI_ATR_ENABLED        0x1L
#define XI_ATR_EDITMENU       0x2L
#define XI_ATR_AUTOSELECT     0x4L
#define XI_ATR_AUTOSCROLL     0x8L
#define XI_ATR_RJUST          0x10L
#define XI_ATR_BORDER         0x20L
#define XI_ATR_VISIBLE        0x40L
#define XI_ATR_FOCUSBORDER    0x80L
#define XI_ATR_READONLY       0x100L
#define XI_ATR_NAVIGATE       0x200L
#define XI_ATR_TABWRAP        0x400L
#define XI_ATR_PASSWORD       0x800L
#define XI_ATR_SELECTED       0x1000L
#define XI_ATR_VCENTER        0x2000L
#define XI_ATR_SELECTABLE     0x4000L
#define XI_ATR_COL_SELECTABLE 0x8000L
#define XI_ATR_HCENTER        0x10000L
```

Description

XI_ATR_ENABLED: If this attribute bit is not set, then the control cannot be operated, and cannot get the focus.

XI_ATR_EDITMENU: This attribute determines whether or not the user can cut, copy and paste to and from the edit field or cell. The **edit_menu** value in the interface definition must also be set to true in order for this to work.

XI_ATR_AUTOSELECT: If this bit is set for types **XIT_COLUMN** or **XIT_FIELD**, then the text of the edit field or cell will be selected when it gains the focus. When the user begins to type, the selected text is replaced.

XI_ATR_AUTOSCROLL: If this attribute is set, then the cell or edit field will automatically scroll horizontally as the user types characters beyond the displayed size.

XI_ATR_RJUST: If this bit is set for types **XIT_COLUMN**, **XIT_FIELD**, or **XIT_TEXT**, then the text in the cell, edit field, or static text control will be right justified.

XI_ATR_BORDER: If this attribute is set, then the edit field will have a border.

XI_ATR_VISIBLE: If this attribute bit is not set, then the object will be invisible (and also disabled).

XI_ATR_FOCUSBORDER: If this attribute is set, then when the edit field or cell has the focus, it will have a border. Typically, **XI_ATR_BORDER** would not be set if this attribute is used.

XI_ATR_READONLY: If this attribute bit is set, then the object cannot be changed. **XIE_ON_*** and **XIE_OFF_*** events are still generated, and double-clicking on object types **XIT_CELL** and **XIT_FIELD** generate **XIE_DBL_CELL** and **XIE_DBL_FIELD** events. Make the object disabled if you do not want these events.

XI_ATR_NAVIGATE: When this attribute is set for an object of type **XIT_LIST**, then navigation with the arrow keys between cells works without the control key pressed, and there is no way to move to the right or left within a cell with the arrow keys. If this attribute is not set then the right and left arrow keys move within a cell, and pressing arrow keys with the control key pressed moves between cells. On the Macintosh, pressing arrow keys with the command key pressed moves between cells.

XI_ATR_TABWRAP: When this attribute is set, and the user presses the tab key at the end of a row on a list, then the focus will proceed to the beginning of the next row. When this attribute is not set, the focus will move to the first cell in the same row.

XI_ATR_PASSWORD: When this attribute is set for type **XIT_COLUMN** or **XIT_FIELD**, the user may type into the edit field, but only cross hatch characters (#) are displayed.

XI_ATR_SELECTED: When this attribute is set, columns or rows are selected and are displayed in reverse video.

XI_ATR_VCENTER: When this attribute is set, text is centered vertically. This attribute is only used internally in XI.

XI_ATR_SELECTABLE: When this attribute is set for a column, the user can select rows by clicking on cells in the column.

XI_ATR_COL_SELECTABLE: When this attribute is set for a column, the user can select the column by clicking on the column heading.

XI_ATR_HCENTER: When this attribute is set for a cell, the text will be centered horizontally. The text will not be centered when editing, so you may want to use this only on disabled cells.

The following is a table of object types and the attributes that apply to them.

XI_BTN	XI_ATR_ENABLED XI_ATR_VISIBLE
XIT_CELL	XI_ATR_RJUST XI_ATR_SELECTED XI_ATR_HCENTER
XIT_COLUMN	XI_ATR_ENABLED XI_ATR_EDITMENU XI_ATR_AUTOSELECT XI_ATR_AUTOSCROLL XI_ATR_FOCUSBORDER XI_ATR_RJUST XI_ATR_READONLY XI_ATR_PASSWORD XI_ATR_SELECTED XI_ATR_SELECTABLE XI_ATR_COL_SELECTABLE
XIT_CONTAINER	NONE
XIT_FORM	NONE
XIT_FIELD	XI_ATR_ENABLED XI_ATR_EDITMENU XI_ATR_AUTOSELECT XI_ATR_AUTOSCROLL XI_ATR_FOCUSBORDER XI_ATR_RJUST XI_ATR_READONLY XI_ATR_PASSWORD XI_ATR_BORDER XI_ATR_VISIBLE
XIT_GROUP	NONE
XIT_ITF	NONE
XIT_LINE	XI_ATR_VISIBLE
XIT_LIST	XI_ATR_ENABLED XI_ATR_VISIBLE XI_ATR_NAVIGATE XI_ATR_TABWRAP
XIT_RECT	XI_ATR_VISIBLE
XIT_ROW	XI_ATR_SELECTED
XIT_TEXT	XI_ATR_ENABLED XI_ATR_RJUST XI_ATR_VISIBLE

XI_BITMAP

Summary

```
typedef struct _xi_bitmap
{
  XinBitmap*      xin_bitmap;
  XI_BITMAP_MODE  mode;
  /* The following properties are only used for XI_BITMAP_NORMAL */
  XinColor        background;
  short           x_offset;
  short           y_offset;
  BOOLEAN         hcenter;
  BOOLEAN         vcenter;
} XI_BITMAP;
```

Description

This structure defines a bitmap that can be assigned to a interface, rectangle, button, or cell. This structure is provided for information. The following macros are provided for modifying the structure:

```
xi_bitmap_background_set
xi_bitmap_hcenter_set
xi_bitmap_mode_set
xi_bitmap_offset_x_set
xi_bitmap_offset_y_set
xi_bitmap_vcenter_set
```

xin_bitmap is a pointer to the bitmap, see **xi_bitmap_create**.

mode is the mode the bitmap will be drawn in. See **XI_BITMAP_MODE** for a description of the allowed values.

background is the color that will be used to fill the background of the rectangle when **XI_BITMAP_NORMAL** is used.

x_offset the number of pixels the bitmap will be offset, within the rectangle, from the upper right in the x direction.

y_offset the number of pixels the bitmap will be offset, within the rectangle, from the upper right in the y direction.

hcenter determines if the bitmap is horizonally centered.

vcenter determines if the bitmap is vertically centered.

XI_BITMAP_MODE Bitmap Modes

Summary

```
typedef enum _xi_bitmap_mode
{
  XI_BITMAP_NORMAL,
  XI_BITMAP_RESIZE,
  XI_BITMAP_TILE
} XI_BITMAP_MODE;
```

Description

This enum defines the mode the bitmap is drawn in.

XI_BITMAP_NORMAL: The bitmap is drawn normal size.

XI_BITMAP_RESIZE: The bitmap is resized to fit the rectangle.

XI_BITMAP_TILE: The bitmap is drawn normal size, multiple times to fit the rectangle.

XI_BTN_DEF Button Object Definition

Summary

```
typedef struct _xi_btn_def
{
  XI_BTN_TYPE type;
  XinRect     xi_rct;
  XinRect     pixel_rect;
  unsigned long attrib;
  char*       text;
  int         tab_cid;
  BOOLEAN     dflt;
  BOOLEAN     cancel;
  int         down_icon_rid;
  int         up_icon_rid;
  int         disabled_icon_rid;
  short       icon_x;
  short       icon_y;
  XI_BITMAP*  down_bitmap;
  XI_BITMAP*  up_bitmap;
  XI_BITMAP*  disabled_bitmap;
  BOOLEAN     checked;
  XinColor    fore_color;
  XinColor    light_color;
  XinColor    ctrl_color;
  XinColor    dark_color;
```

```
  BOOLEAN     drawable;
  XI_BTN_DRAW draw_as;
  char        mnemonic;
  short       mnemonic_instance;
  XVT_FNTID font_id;
} XI_BTN_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. Use this structure if you are modifying an existing button definition. Otherwise, use the convenience function **xi_add_button_def**, which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

type determines the type of button to create. See **XI_BTN_TYPE** for the list of possible values.

xi_rct is the bounding rectangle for the button control, in form units. This rectangle is ignored if the **pixel_rct** values are set.

pixel_rect is the bounding rectangle for the button control, in pixels.

attrib is a bitwise OR'ed combination of **XI_ATR_*** values which control some of the behavior of the button. The following attributes apply to buttons:

```
XI_ATR_ENABLED
XI_ATR_VISIBLE
```

text is the label of the button.

tab_cid is the control that is next in the tabbing sequence. Buttons should only tab to a control that can actually have focus.

dflt determines whether the button has the look and feel of the default button. The default button will be pressed when the user presses the enter key, unless some other button has focus. The enter key also behaves differently if the **tab_on_enter** value is set to **TRUE** in the interface definition. **dlft** should only be set for pushbuttons, not radio buttons, check boxes, or tab buttons.

cancel determines whether the button has the look and feel of the cancel button.

down_icon_rid, **up_icon_rid**, and **disabled_icon_rid** define the resource ID's for icon buttons. Icons resource ID's should be set only for pushbuttons, not for radio buttons, check boxes, or tab buttons.

icon_x and **icon_y** define the distance from the upper left of the button, in pixels, to display the various icons for icon buttons.

down_bitmap, up_bitmap, and disabled_bitmap are bitmaps that will be drawn on the pushbutton. Different bitmaps can be supplied for the three states. The **up_bitmap** will be used if **down_bitmap** or **disabled_bitmap** is not supplied. You must create these bitmaps using **xi_create_bitmap**, the local copies of the bitmaps are destroyed by calling **xi_def_free**, after the call to xi_create.

checked defines the initial state for radio buttons, check boxes, and tab buttons. This includes checkboxes and radio buttons that have a pushbutton appearance. Only one of a container of radio buttons or tab buttons should be checked initially.

fore_color defines the foreground color for radio buttons and check boxes and the color of text in pushbuttons and tab buttons. Sometimes it might be desirable to have several radio buttons be a different color than black. You might, for instance, have a group of red radio buttons and a group of green radio buttons.

light_color, ctrl_color and **dark_color** override the **XI_PREF_COLOR_LIGHT**, **XI_PREF_COLOR_CTRL** and **XI_PREF_COLOR_DARK** setting for a particular button.

If **drawable** is set to **TRUE**, then the application can draw into a push button. Any time that the push button changes state, such as when it is depressed with the mouse, XI generates an **XIE_UPDATE** event. This option should only be set for push buttons, not for radio buttons, check boxes, or tab buttons. **XI_PREF_NATIVE_CTRLS** must be set to **FALSE** for this option to work.

draw_as determines if the button is drawn as a native control, or emulated.

mnemonic defines the character, used in combination with the ALT key, that will press the button.

mnemonic_instance defines which instance of the mnemonic character to underline. Setting this to 1 will cause the first instance of the mnemonic to be underlined.

font_id value is used to override the font from the interface. You are responsible for destroying the XVT R4 font, which you can do after the call to xi_create. If you call xi_def_free to destroy the definition the font_id will be destroyed for you.

XI_BTN_DRAW Button

Summary

```
typedef enum _xi_btn_draw
{
  XIBT_DEFAULT,
  XIBT_EMULATED,
  XIBT_NATIVE
} XI_BTN_DRAW;
```

Description

This enum defines the methods of button creation that can be used.

XIBT_DEFAULT: This takes the value of XI_PREF_NATIVE_CTRLS.

XIBT_EMULATED: This causes XI to draw the control.

XIBT_NATIVE: This causes XI to use the native control.

XI_BTN_TYPE Button Types

Summary

```
typedef enum _xi_btn_type
{
  XIBT_BUTTON,
  XIBT_CHECKBOX,
  XIBT_RADIOBTN,
  XIBT_TABBTN,
```

```
  XIBT_BUTTON_CHECKBOX,
  XIBT_BUTTON_RADIOBTN
} XI_BTN_TYPE;
```

Description

This enum defines the types of buttons that can be created.

XIBT_BUTTON: This is a standard pushbutton.

XIBT_CHECKBOX: This is a standard checkbox.

XIBT_RADIOBTN: This is a standard radio button.

XIBT_TABBTN: This is a radio button that is drawn to look like a file folder tab. It is made to fit above a **XIT_RECT** control.

XIBT_BUTTON_CHECKBOX: This is a checkbox that looks like a pushbutton. The button will stay depressed when it is checked.

XIBT_BUTTON_RADIOBTN: This is a radio button that looks like a pushbutton. The button will stay depressed when it is checked.

XI_CELL_DATA Cell Data Specification

Summary

```
typedef struct _xi_cell_data
{
  unsigned char row;
  unsigned char column;
  unsigned char is_vert_scrolled;
} XI_CELL_DATA;
```

Description

This is used to specify a cell object.

row is the row number of the cell. This row number refers to its position in the visible part of the list. If you want the record handle for that row, use **xi_get_list_info** to get the record handle array and use **row** to index that array.

column is the column number of the cell. If you want the particular column information, use this value to index the **children** member of the **XI_OBJ** structure for the list.

is_vert_scrolled will be **TRUE** if the row with this cell is not currently visible. This will only happen if you are using the **XI_PREF_KEEP_FOCUS_FIXED** option and the cell is on the focus row when it is scrolled off the list vertically.

See Also

XI_MAKE_CELL, XI_OBJ

XI_CELL_SPEC Cell Specification

Summary

```
typedef struct
{
  short row;
  short column;
} XI_CELL_SPEC;
```

Description

This structure is used by **xi_get_cell_selection**, which returns an array of these structures. Each element in the array is a cell that is currently selected. See **xi_get_cell_selection** for further details.

XI_COLOR_PART Color Part Specification

Summary

```
typedef enum
{
  XIC_ENABLED,
  XIC_BACK,
  XIC_HILIGHT,
  XIC_SHADOW,
  XIC_ACTIVE,
  XIC_DISABLED,
  XIC_DISABLED_BACK,
  XIC_WHITE_SPACE,
  XIC_ACTIVE_BACK
} XI_COLOR_PART;
```

Description

This enum is used by **xi_set_color**. Color may be set separately for the various parts of an XI object. See **xi_set_color** for further details.

XI_COLUMN_DEF Column Object Definition

Summary

```
typedef struct _xi_column_def
{
  unsigned long attrib;
  short sort_number;
  short width;
  short pixel_width;
  short text_size;
  char *heading_text;
  BOOLEAN center_heading;
  BOOLEAN heading_well;
  BOOLEAN heading_platform;
  BOOLEAN column_well;
  BOOLEAN column_platform;
  short position;
  int icon_rid;
  short icon_x;
  short icon_y;
  XI_BITMAP* bitmap;
  BOOLEAN size_rows;
  BOOLEAN suppress_update_heading;
  BOOLEAN suppress_update_cells;
  BOOLEAN vertical_align_center;
  BOOLEAN vertical_align_bottom;
  BOOLEAN wrap_text;
  BOOLEAN wrap_text_scrollbar;
  BOOLEAN auto_tab;
  BOOLEAN cr_ok;
  BOOLEAN var_len_text;
  char mnemonic
  XI_ICON_MODE_TYPE icon_mode;
  XVT_FNTID font_id;
} XI_COLUMN_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. Use this structure only if you are modifying an existing column definition. Otherwise, use the convenience function **xi_add_column_def**, which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

attrib is an OR'ed combination of **XI_ATR_*** which controls some of the behavior of the column. The following attributes apply to columns:

```
XI_ATR_ENABLED
XI_ATR_EDITMENU
XI_ATR_AUTOSELECT
XI_ATR_AUTOSCROLL
XI_ATR_FOCUSBORDER
XI_ATR_RJUST
XI_ATR_READONLY
XI_ATR_PASSWORD
XI_ATR_SELECTED
XI_ATR_SELECTABLE
XI_ATR_COL_SELECTABLE
```

sort_number is used by **xi_create** when adding a column to an existing list. The new column created by **xi_create** will be inserted in the list such that all the columns will be arranged from left to right based on ascending values of their **sort_number**. In general, it is easier to use the **position** value.

width is the width of the column, in form units. This value is ignored if **pixel_width** is specified.

pixel_width is the width of the column, in pixels.

text_size specifies the size of the edit buffer for cells in the column. If the **XI_ATR_AUTOSCROLL** attribute is set and **text_size** specifies more characters than will be visible in the cell, then the text will scroll horizontally to display the entire edit buffer. If the **var_len_text** flag is set, this size is used at creation and may be overrode later.

icon_mode determines how a icon and text interact when both are specified.

If **icon_rid** is non-zero, then the icon specified is placed in the column heading. **icon_x** and **icon_y** specify an x and y offset from the upper left corner of the column heading, for drawing the icon.

If **bitmap** is set, then the bitmap specified is placed in the column heading. The bitmap is created by calling **xi_bitmap_create**. Calling **xi_def_free** on the definition, after calling **xi_create** will free up the local pointer.

heading_text is the text for the heading of a column. If the list has no headings, this value is ignored.

If **center_heading** is set, then the column heading text will be centered.

If **heading_well** is set, then the column heading will appear 3D, and be indented. If **heading_platform** is set, then the column heading will appear 3D, and will be a platform. Only one of **heading_well** and **heading_platform** may be set for a column definition.

If **column_well** is set, then the column cells will appear 3D, and be indented. If **column_platform** is set, then the column cells will appear 3D, and will have the appearance of a platform. Only one of **column_well** and **column_platform** may be set for a column definition.

When inserting a column into an existing list, **position** determines the position to insert the new column. If **position** is set, then **sort_number** is ignored.

The **font_id** value is used to override the font from the interface. You are responsible for destroying the XVT R4 font, which you can do after the call to xi_create.

If **size_rows** is set to **TRUE**, then the user can change the height of a row by dragging on the border between the rows. When the user sizes a row, XI generates an **XIE_ROW_SIZE** event. The application can remember the height of the row by saving the **new_row_height** field upon the **XIE_ROW_SIZE** event. Then, when the application receives a record request event for the row, the application can set the **row_height** field in the **XI_EVENT** structure.

If **suppress_update_heading** is **TRUE**, XI will not draw anything in the heading area for this column. You are responsible for drawing everything in that area during the **XIE_UPDATE** event. You can determine the size of that area by calling **xi_get_rect** for the column object which gives you the entire column area, including the cells. You can then call **xi_get_rect** for the first cell or the first row and the top of that area will be the bottom of the column heading.

If **suppress_update_cells** is **TRUE**, XI will not draw anything in the cell area for this column. You are responsible for drawing everything in that area during the **XIE_UPDATE** event. You can use **xi_get_rect** on one or more cells to determine the area to update. Alternatively, you can call **xi_get_rect** for the column, which will include the heading, and then call **xi_get_rect** for the first cell which gives you the top of the cell area.

If **vertical_align_center** is **TRUE**, then the text will be centered vertically in rows that are taller than required to display that text.

If **vertical_align_bottom** is **TRUE**, then the text will be displayed at the bottom of rows that are taller than required to display that text.

If **wrap_text** is **TRUE**, text in cells for the column will be wrapped to multiple lines up to the value set for **XI_PREF_DEFAULT_MAX_LINES_IN_CELL**. Editing of text in the cell is still done on a single line with horizontal scrolling.

If **wrap_text_scrollbar** is TRUE, and wrap_text is TRUE, a vertical scrollbar will appear when the cell gains focus.

If **auto_tab** is set to **TRUE**, the focus will leave the cell when it is full, as if the tab key was pressed. The cell is full when it has **text_size** characters in it. This feature is provided to support “heads-down” data entry.

If **cr_ok** is set, and the cell is wrapped, a carriage return will be directed to the cell, rather than press a default button.

mnemonic defines the character, used in combination with the ALT key, that will press the button.

If **var_len_text** is TRUE the initial buffer size is what was defined at creation. The application can reallocate the string to the size needed by using xi_tree_realloc in the XIE_CELL_REQUEST event. The internal buffer is automatically expanded and decreased as the user types information into each cell. When getting the text, call xi_get_text with a string of NULL, uses strlen to get the size of the internal string, then call xi_get_text again, with the correct length parameter.

XI_CONTAINER_DEF Container Object Definition

Summary

```
typedef struct _xi_container_def
{
  XinRect   xi_rct;
  XinRect   pixel_rect;
  XI_CONTAINER_ORIENTATION orientation;
  int      tab_cid;
  short    btn_height;
  short    btn_width;
  BOOLEAN  packed
} XI_CONTAINER_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure and it is also passed to the **xi_container_reorient** function. Use this structure if you are modifying an existing container definition. Otherwise, use the convenience function **xi_add_container_def**, which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

xi_rct defines the rectangle, in form units, that will contain all of the buttons. This rectangle is ignored if the **pixel_rect** values are set.

pixel_rect is the rectangle, in pixels, that will contain all of the buttons.

orientation defines whether the buttons are stacked horizontally, vertically, or in a grid. See **XI_CONTAINER_ORIENTATION** for more information.

tab_cid is the control id of the form, list, or container that is next in the meta-tabbing sequence. The focus will go to a control within that object when the character defined in the preference **XI_PREF_ITF_TAB_CHAR** is pressed.

If the buttons are stacked in a grid, then **btn_height** and **btn_width** define the height and width of the buttons in the grid. If either **btn_height** or **btn_width** are zero, then the buttons in the grid are made square.

If **packed** is set to **TRUE**, then the buttons in the container are packed closely together. In this case, there is no room around the button for the rectangle that indicates focus. In addition, there is no room for the rectangle that indicates the default button. This option should only be used for buttons that will be pressed with the mouse. The application should not include packed buttons in the tabbing sequence. When buttons are packed, the width of the grid lines between the buttons is set by the preference **XI_PREF_CONTAINER_GRID_WIDTH**.

See Also

xi_container_reorient

XI_CONTAINER_ORIENTATION Container Orientation

Summary

```
typedef enum
{
  XI_STACK_HORIZONTAL,
  XI_STACK_VERTICAL,
  XI_GRID_HORIZONTAL,
  XI_GRID_VERTICAL
} XI_CONTAINER_ORIENTATION;
```

Description

This enum is used in conjunction with **XI_CONTAINER_DEF**. It defines whether the buttons contained in a container are stacked horizontally, vertically, or in a grid.

If buttons are stacked in a grid, they can be arranged horizontally or vertically. For example, if the buttons in the grid are arranged horizontally, then the second button in the container will be to the right of the first button. If the buttons in the grid are arranged vertically, then the second button in the container will be below the first button.

XI_CURSOR_* Cursor Resource ID

Summary

```
#define XI_CURSOR_RESIZE  8001
#define XI_CURSOR_HAND    8002
#define XI_CURSOR_VRESIZE 8003
```

Description

These define the resource ID's that are used for special cursors in XI. The "resize" cursor appears at the line between column headings on a list that has resizable columns. The "hand" cursor appears over the column headings on a list that has movable columns. The "vertical resize" cursor appears at the line between rows on a list with sizable rows.

These are the default values for the cursor resource ID's. They can be set to something else using **xi_set_pref**.

If you are going to use any of these options on the list, the appropriate cursors must appear in your URL file. The URL for the example programs demonstrate how to do this.

See Also

XI_PREF_SIZE_CURSOR_RID, XI_PREF_HAND_CURSOR_RID, XI_PREF_VSIZE_CURSOR_RID, xi_set_pref

XI_EVENT Event Information

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    EVENT xvte;
    struct _xi_obj *xi_obj;
    struct xit_rec_request
    {
      struct _xi_obj *list;
      long spec_rec;
      long data_rec;
      short percent;
      unsigned long attrib;
      XinColor color;
      int row_height;
      BOOLEAN has_focus;
    } rec_request;
    struct xit_rec_allocate
    {
      struct _xi_obj *list;
      long record;
    } rec_allocate;
    struct xit_rec_free
    {
      struct _xi_obj *list;
      long record;
    } rec_free;
    struct xit_get_percent
    {
      struct _xi_obj *list;
      long record;
      short percent;
    } get_percent;
```

```
struct xit_select
{
  struct _xi_obj *xi_obj;
  BOOLEAN selected;
  BOOLEAN dbl_click;
  BOOLEAN shift;
  BOOLEAN control;
  int column;
} select;
struct xit_cell_request
{
  struct _xi_obj *list;
  char *s;
  short len;
  long rec;
  short col_nbr;
  int icon_rid;
  unsigned long attrib;
  XinColor color;
  XinColor back_color;
  XVT_FNTID font_id;
  BOOLEAN button;
  BOOLEAN button_on_left;
  BOOLEAN button_on_focus;
  int button_icon_rid;
} cell_request;
struct xit_row_size
{
  struct _xi_obj *xi_obj;
  int new_row_height;
} row_size;
struct xit_cmd
{
  int tag;
  BOOLEAN shift;
  BOOLEAN control;
} cmd;
struct xit_chr
{
  struct _xi_obj *xi_obj;
  int ch;
  BOOLEAN shift;
  BOOLEAN control;
  BOOLEAN is_paste;
} chr;
struct xit_vir_pan
{
  BOOLEAN before_pan;
  int delta_x;
  int delta_y;
} vir_pan;
```

```
    struct xit_column
    {
      struct _xi_obj *list;
      int   col_nbr;
      int   new_col_nbr;
      BOOLEAN in_fixed;
      int   new_col_width;
    } column;
    long user_data;
  } v;
} XI_EVENT;
```

Description

This structure contains the data that will be sent to the XI event handler function when an **XI_EVENT** is generated. All references to the structure **_xi_obj** are forward references to the **XI_OBJ** structure.

Note: **refused** is set to **FALSE** before entry to an XI event handler.

The *XI Events* chapter documents the specific members of this structure for each event.

user_data can be used for your own events that you wish to pass to the XI event handler. If you want to create your own events, use an event type greater than **XI_MAX_EVENT**.

XI_EVENT_HANDLER XI Event Handler

Summary

```
typedef void (*XI_EVENT_HANDLER)( XI_OBJ *itf, XI_EVENT *xiev);
```

Description

XI_EVENT_HANDLER is the type definition for a pointer to a event handler function. This event handler function is associated with each XI interface when it is created.

NOTE: The typedef shown above is approximate. It sometimes varies to accommodate variations in compilers on the different platforms.

See Also

XI_EVENT, XI_EVENT_TYPE and the *XI Events* chapter.

XI_EVENT_TYPE Event Type

Summary

```
typedef enum _xi_event_type
{
  XIE_CHAR_FIELD,
  XIE_DBL_FIELD,
  XIE_CHG_FIELD,
  XIE_OFF_FIELD,
  XIE_ON_FIELD,
  XIE_OFF_GROUP,
  XIE_ON_GROUP,
  XIE_OFF_FORM,
  XIE_ON_FORM,
  XIE_VIR_PAN,
  XIE_XVT_EVENT,
  XIE_XVT_POST_EVENT,
  XIE_INIT,
  XIE_BUTTON,
  XIE_CHAR_CELL,
  XIE_CLEANUP,
  XIE_CLOSE,
  XIE_COMMAND,
  XIE_DBL_CELL,
  XIE_GET_FIRST,
  XIE_GET_LAST,
  XIE_GET_NEXT,
  XIE_GET_PERCENT,
  XIE_GET_PREV,
  XIE_CELL_REQUEST,
  XIE_CHG_CELL,
  XIE_OFF_CELL,
  XIE_ON_CELL,
  XIE_OFF_ROW,
  XIE_ON_ROW,
  XIE_OFF_COLUMN,
  XIE_ON_COLUMN,
  XIE_OFF_LIST,
  XIE_ON_LIST,
  XIE_REC_ALLOCATE,
  XIE_REC_FREE,
  XIE_ROW_SIZE,
  XIE_SELECT,
  XIE_UPDATE,
  XIE_COL_DELETE,
  XIE_COL_MOVE,
  XIE_COL_SIZE
} XI_EVENT_TYPE;
```

Description

This enum defines the types of XI events. In your application's event handler function, you would switch off of the **type** field in the **XI_EVENT** structure and take appropriate action for each of the XI events.

The details of each event are documented in the *XI Event* chapter.

See Also

XI_EVENT

XI_FIELD_DEF Edit Field Object Definition

```
typedef struct _xi_field_def
{
  XinPoint pnt;
  short    field_width;
  XinPoint pixel_origin;
  short    pixel_width;
  short    pixel_button_distance;
  XinRect  xi_rct;
  unsigned long attrib;
  int   tab_cid;
  short text_size;
  XinColor back_color;
  XinColor enabled_color;
  XinColor disabled_color;
  XinColor disabled_back_color;
  XinColor active_color;
  XinColor active_back_color;
  XinColor hilight_color;
  XinColor shadow_color;
  BOOLEAN button;
  BOOLEAN button_on_left;
  int icon_rid;
  XI_BITMAP* button_bitmap;
  BOOLEAN well;
  BOOLEAN platform;
  BOOLEAN auto_tab;
  BOOLEAN scroll_bar;
  BOOLEAN cr_ok;
  char mnemonic;
  BOOLEAN var_len_text;
  XVT_FNTID font_id;
} XI_FIELD_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. Use this structure only if you are modifying an existing edit field definition. Otherwise, use the convenience function **xi_add_field_def**, which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

pnt is the upper-left corner of the edit field, in form units. This point is ignored if **pixel_origin** is specified.

field_width is the width of the edit field, in form units. This value is ignored if **pixel_width** is specified.

pixel_origin is the upper-left corner of the edit field, in pixels.

pixel_width is the width of the edit field, in pixels.

pixel_button_distance specifies the number of pixels between the edge of the edit field and the field button. This value is ignored if there is no field button. If this value is not specified, the outside edge of the field button will be aligned to a form unit boundary.

xi_rct specifies the bounding rectangle, in form units, for the edit field. If this rectangle is more than eight form units high, then XI create a multi-line edit field. **NOTE**: Multi-line edit fields are not currently supported for virtual interfaces.

attrib is an OR'ed combination of **XI_ATR_*** values which control some of the behavior of the edit field.The following attributes apply to edit fields:

```
XI_ATR_ENABLED
XI_ATR_EDITMENU
XI_ATR_AUTOSELECT
XI_ATR_AUTOSCROLL
XI_ATR_FOCUSBORDER
XI_ATR_RJUST
XI_ATR_READONLY
XI_ATR_PASSWORD
XI_ATR_BORDER
XI_ATR_VISIBLE
```

tab_cid is the control that is next in the tabbing sequence.

text_size specifies the size of the edit buffer in the edit field. If the **XI_ATR_AUTOSCROLL** attribute is set and **text_size** specifies more characters than will be visible in the edit field, then the text will scroll horizontally to display the entire edit buffer. On a multi-line edit field, this value specifies the paragraph limit for single paragraph multi-line edit fields, unless **var_len_text** is set.

back_color, **enabled_color**, **disabled_color**, **disabled_back_color**, **active_color** and **active_back_color** specify the colors to be used when the edit field changes states, in order to enhance feedback to the user. The enabled, disabled, and active colors are all foreground colors for the text, and the pen color for the border. The "back" colors are background colors for enabled, disabled and active states. The **back_color** is ignored on 3D edit fields, unless **hilight_color** and **shadow_color** are specified.

hilight_color and **shadow_color** are used for well and platform edit fields. It is possible to use different 3D shading for each edit field. Well and platform edit fields ignore the **back_color** field, unless **hilight_color** and **shadow_color** are set.

button determines whether XI will place an edit field button with a down arrow in it to the right or left of the edit field. If **button_on_left** is set to **TRUE**, then the edit field button will be placed to the left of the edit field. If **button_on_left** is set to **FALSE**, then the edit field button will be placed to the right of the edit field. **icon_rid** specifies the icon resource id of the icon to be placed in the edit field button. Or, **bitmap** specifies a bitmap to be placed in the edit field button. It is created by calling **xi_create_bitmap**, and destroyed by calling **xi_def_free** on the definition, after calling **xi_create**.

If **well** is set, then the edit field will have a 3D appearance, and have an "indented" appearance.

If **platform** is set, then the edit field will have a 3D appearance, and have a "raised" appearance.

Only one of **well** or **platform** may be set for an edit field.

If **auto_tab** is set to **TRUE**, the focus will leave the edit field when it is full, as if the tab key was pressed. The edit field is full when it has **text_size** characters in it. This feature is provided to support "heads-down" data entry.

If **scroll_bar** is set, and the field is multi-line, it will have a vertical scrollbar when it gains focus.

If **cr_ok** is set, and the field is multi-line, a carriage return will be directed to the field, rather than press a default button.

mnemonic defines the character, used in combination with the ALT key, that will press the button.

If **var_len_text** is TRUE edit fields are not constrained by any buffer size. The internal buffer is automatically expanded and decreased as the user types information into the field. When getting the text, call xi_get_text with a string of NULL, uses strlen to get the size of the internal string, then call xi_get_text again with the correct length parameter.

The **font_id** value is used to override the font from the interface. You are responsible for destroying the XVT R4 font, which you can do after the call to xi_create.

XI_FORM_DEF Form Object Definition

Summary

```
typedef struct _xi_form_def
{
  int tab_cid;
} XI_FORM_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. In general, you will not need to access this definition directly, instead use the convenience function **xi_add_form_def** which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

tab_cid is the control id of the form, list, or container that is next in the meta-tabbing sequence. The focus will go to a control within that object when the character defined in the preference **XI_PREF_ITF_TAB_CHAR** is pressed.

XI_FU_MULTIPLE Form Units Per Character

Summary

```
#define XI_FU_MULTIPLE 8
```

Description

Each form unit is 1/8th the width of a standard character or 1/8th the height of a standard edit field. You can use **XI_FU_MULTIPLE** to reflect this relationship. For example, the first edit field in a window may have a top position of **XI_FU_MULTIPLE**. The edit field directly below it will have a top position of 2 * **XI_FU_MULTIPLE**.

XI_GROUP_DEF Group Object Definition

Summary

```
typedef struct _xi_group_def
{
  short nbr_cids;
  int *cids;
} XI_GROUP_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. In general, you will not need to access this definition directly, instead use the convenience function **xi_add_group_def** which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

Whenever the focus enters or leaves the set of controls in the group, then a **XIE_ON_GROUP** or **XIE_OFF_GROUP** event will be generated.

nbr_cids and **cids** define the list of objects that will be associated with a group. **cids** is an array of integers that contain the control ID's of the objects in the group. These objects associated with the control ID's in **cids** must be of types **XIT_FIELD**, or **XIT_COLUMN**.

Groups only make sense if a group is composed entirely of columns, or entirely of edit fields. If a group is made up of columns, then a group is effectively created for each row of the list, so that moving from one row to the next can cause an **XIE_OFF_GROUP** event.

XI_ICON_MODE_TYPE

Summary

```
typedef enum _xi_icon_mode_type
```

```
{
  XIM_ICON_HAS_PRIORITY,
  XIM_TEXT_WITH_ICON_ON_LEFT,
  XIM_TEXT_WITH_ICON_ON_RIGHT
}
```

Description

This enum defines the modes the heading icon and text can be drawn in.

XIM_ICON_HAS_PRIORITY: If a icon and text are both specified, just the icon is drawn.

XIM_TEXT_WITH_ICON_ON_LEFT: Both a icon and text are drawn. The icon is to the left of the text.

XIM_TEXT_WITH_ICON_ON_RIGHT: Both a icon and text are drawn. The icon is to the right of the text.

XI_INTERNAL Exposes Internal Structures

Summary

```
#define XI_INTERNAL
#include "xi.h"
```

Description

By defining **XI_INTERNAL** before the include of "xi.h", you can then access the internal members of the **XI_OBJ** structure. This should only be done if you cannot get the information you need through other XI functions.

WARNING: Most of the internal structure of **XI_OBJ** is undocumented and subject to change without notice.

See Also

XI_OBJ

XI_ITF_DEF Interface Object Definition

Summary

```
typedef struct _xi_itf_def
{
  XI_EVENT_HANDLER xi_eh;
  XinRect *rctp;
  char *title;
```

```
  BOOLEAN ctl_size;
  BOOLEAN ctl_vscroll;
  BOOLEAN ctl_hscroll;
  BOOLEAN ctl_close;
  BOOLEAN ctl_iconized;
  BOOLEAN ctl_iconizable;
  int menu_bar_rid;
  XinWindow win;
  XinWindow menu_win;
  BOOLEAN size_win;             /* automatically size window win */
  BOOLEAN edit_menu;
  XinColor back_color;
  BOOLEAN automatic_back_color;
  unsigned long attrib;
  BOOLEAN virtual_itf;
  BOOLEAN modal;
  BOOLEAN size_font_to_win;
  BOOLEAN tab_on_enter;
  BOOLEAN use_whitespace;
  int whitespace_right;
  int whitespace_bottom;
  BOOLEAN use_xil_win;
  BOOLEAN autoclose;
  XI_BITMAP* bitmap;
  BOOLEAN modal_wait;
  BOOLEAN border_style_set;
  XinBorderStyle border_style;
  BOOLEAN visible;
  BOOLEAN enabled;
  BOOLEAN maximized;
  XinWindow parent;
  XinMenu *menu;
  int icon_rid;
  XVT_FNTID font_id;
  int initial_focus_cid;
} XI_ITF_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. Use this structure only if you are modifying an existing interface definition. Otherwise, use the convenience function **xi_create_itf_def**, which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

xi_eh is a pointer to the event handler function that will receive XI events for this interface. Multiple interfaces can share the same event handler function.

If **win** is specified, then XI will create the interface in that existing XVT window. Otherwise, XI will create an XVT window for the interface. If XI creates the window, it will size that window appropriately for the controls. If XI does not create the window, it will only set the window size if **size_win** is set to **TRUE**. The following options apply only if XI creates the XVT window:

rctp defines the size of the window to be used for the interface. If **rctp** is **NULL** or if the height and width are zero, then XI will compute a window large enough to hold the interface. In addition, if the application does not pass **NULL** for **rctp**, and the height and width are zero, then the window will be placed at **rctp->top** and **rctp->left**. Otherwise, the application can call **xi_get_def_rect** to determine the bounding rectangle for the interface, and then enlarge the rectangle in order to place more information or controls on the window. You can set this pointer to point to a local **XI_RCT** variable as long as you call **xi_create** before the end of the function.

title will be the title in the window created for the interface. You can set this pointer to a local variable as long as you call **xi_create** before the end of the function.

If **ctl_size** is **TRUE**, then the window will have the appropriate sizing controls for that platform.

If **ctl_vscroll** is **TRUE**, then the window will have a vertical scroll bar.

If **ctl_hscroll** is **TRUE**, then the window will have a horizontal scroll bar.

If **ctl_close** is **TRUE**, then the window will have a close box.

If **ctl_iconized** is **TRUE**, then the window will be created iconized.

If **ctl_iconizable** is **TRUE**, then the window will have an iconize box.

If **menu_bar_rid** is set, then XI creates the window with the specified menu bar.

If **modal** is set to **TRUE**, then the window is created with some modal behavior. This window will stay on top and all other windows in the application will be disabled, however, the call to **xi_create** will still return immediately rather than waiting until the window closes.

Modal windows **must** be closed using **xi_delete**, or by not refusing the **XIE_CLOSE** event. XI modal windows can be nested and any window, modal or otherwise, can be closed with **xi_delete** while those modal windows are open.

If **modal_wait** is set to **TRUE**, then the window is created with true modal behavior. The call to **xi_create** will not return until the window closes. This applies to applications running on XVT 4.5 or higher only.

border_style and **border_style_set** allow you to specify the type of window to create, see **XinBorderStyle** for more detail. To get the style specified with **border_style** you must also set **border_style_set** to TRUE.

If **visible** is **FALSE**, the window is created invisible.

If **enabled is FALSE,** the window is created disabled.

If **maximized** is set to **TRUE**, the interface window will be maximized on creation.

parent determines which window the interface is a child of. It is TASK_WIN by default.

menu is not supported at this time.

If **icon_rid** is set**,** and the application is on W16, OS2, or W32, this icon will be used to represent the minimized window.

If **edit_menu** is set to **TRUE**, then XI will modify the state of the edit menu, as the user cuts, pastes, and selects text. If the interface is in a child window, you can set **menu_win** to the parent which has the menu you want updated.

back_color specifies the background color of the interface. It is better to let XI draw the background color, because XI can then do certain optimizations when scrolling rectangular regions.

If **automatic_back_color** is set to **TRUE**, then XI will pick the best color for the background color on the particular platform. The background will generally be a shade of gray, except on character systems, where it is black.

attrib is not used for an interface.

If **virtual_itf** is set to **TRUE**, then XI will allow the developer to create a larger XI interface than the actual XVT window. Then, the user can "pan" across the XVT window to make different parts of the XI interface visible. The window *must* have both a vertical and horizontal scroll bar. If the application created the window before calling **xi_create**, then the application must have created the window with both vertical and horizontal scroll bars. If the application will let XI create the window, then **ctl_hscroll** and **ctl_vscroll** need to be set to **TRUE**.

If **size_font_to_win** is **TRUE**, XI will try to adjust the font size so that all controls will fit into the current window size. The size will adjust dynamically as the window is resized.

If **tab_on_enter** is **TRUE**, the enter key will work like the tab key on any control except pushbuttons. This option is provided to support "heads-down" data entry.

If **use_whitespace** is **TRUE**, then XI will use the **whitespace_right** and **whitespace_bottom** values instead of the current settings for **XI_PREF_ITF_WS_RIGHT** and **XI_PREF_ITF_WS_BOTTOM**.

The **font_id** value is used to override the font from the interface. You are responsible for destroying the XVT R4 font, which you can do after the call to xi_create.

If **use_xil_win** is **TRUE**, any list in the interface will bahave like the XVT PowerObjects version of the XI spreadsheet. This overrides a **FALSE** setting of **XI_PREF_XIL**.

If **autoclose** is TRUE the interface window will close when another window gains focus. This is useful with drop down lists.

bitmap is a pointer to a bitmap, created with **xi_bitmap_create**, that is drawn as background for the interface. The local copy of the bitmap is destroyed by calling **xi_def_free**, after the call to **xi_create**.

initial_focus_cid contains the control id of the object that should have initial focus.

XI_LINE_DEF Line Object Definition

Summary

```
typedef struct _xi_line_def
{
  XinPoint  pnt1;
  XinPoint pnt2;
  XinPoint pixel_pnt1;
  XinPoint pixel_pnt2;
  XinColor fore_color;
  XinColor back_color;
  BOOLEAN well;
  unsigned long attrib;
} XI_LINE_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. Use this structure if you are modifying an existing line definition. Otherwise, use the convenience function **xi_add_line_def**, which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

pnt1 is the starting position of the line, in form units. This value is ignored if **pixel_pnt1** is specified.

pnt2 is the ending position of the line, in form units. This value is ignored if **pixel_pnt2** is specified.

pixel_pnt1 is the starting position of the line, in pixels.

pixel_pnt2 is the ending position of the line, in pixels.

fore_color is the foreground color of the line, if the 3D appearance is not used. **back_color** is the background color of the line, (for XVT/CH only) if the 3D appearance is not used. If the 3D appearance is used, then the colors of the line are determined by the preferences **XI_PREF_COLOR_LIGHT** and **XI_PREF_COLOR_DARK**.

If the 3D appearance is used, then lines either appear as “wells” or “platforms”. If **well** is set to **TRUE**, then the line has the appearance of being indented. If **well** is not set, then the line has the appearance of being raised.

attrib is an OR’ed combination of **XI_ATR_*** values which control the behavior of the line. The following attributes apply to lines:

```
XI_ATR_VISIBLE
```

XI_LIST_DEF List Object Definition

Summary

```
typedef struct _xi_list_def
{
  XinPoint xi_pnt;
  short height;
  short width;
  XinPoint pixel_origin;
  short   pixel_height;
  short   pixel_width;
  unsigned long attrib;
  XinColor back_color;
  XinColor enabled_color;
  XinColor disabled_color;
  XinColor disabled_back_color;
  XinColor active_color;
  XinColor active_back_color;
  XinColor white_space_color;
  XinColor rule_color;
  BOOLEAN no_heading;
  BOOLEAN one_row_list;
  BOOLEAN scroll_bar;
  BOOLEAN sizable_columns;
  BOOLEAN movable_columns;
  BOOLEAN scroll_bar_button;
  short fixed_columns;
  int tab_cid;
  short min_cell_height;
  short min_heading_height;
  BOOLEAN no_horz_lines;
  BOOLEAN no_vert_lines;
  int start_percent;
  int first_vis_column;
  BOOLEAN drop_and_delete;
  BOOLEAN select_cells;
  BOOLEAN get_all_records;
  BOOLEAN resize_with_window;
  int horz_sync_list;
  int vert_sync_list;
  BOOLEAN row_focus_border;
  XinColor row_focus_border_color;
  int max_lines_in_cell;
  BOOLEAN single_select;
  BOOLEAN retain_back_color_on_select;
  BOOLEAN drag_and_drop_rows;
  BOOLEAN drag_rows_autoscroll;
  BOOLEAN button_on_cell_focus;
  XVT_FNTID font_id;
} XI_LIST_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. Use this structure if you are modifying an existing list definition. Otherwise, use the convenience function **xi_add_list_def**, which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

xi_pnt is the upper-left corner of the list, in form units.

height is the height of the list, in form units. XI will adjust the number of rows in the list to fit as many lines as possible into the given height.

If **width** is set to a non-zero number of form units, then the list will become a horizontally scrolling list, and XI will place a horizontal scroll bar at the bottom of the list.

pixel_origin is the upper-left corner of the list, in pixels.

pixel_height is the height of the list, in pixels. XI will adjust the number of rows in the list to fit as many lines as possible into the given height.

If **pixel_width** is set to a non-zero number of pixels, then the list will become a horizontally scrolling list, and XI will place a horizontal scroll bar at the bottom of the list.

attrib is an OR'ed combination of **XI_ATR_*** values which control the behavior of the list. The following attributes apply to list:

```
XI_ATR_ENABLED
XI_ATR_VISIBLE
XI_ATR_NAVIGATE
XI_ATR_TABWRAP
```

back_color, **enabled_color**, **disabled_color, disabled_back_color**, **active_color**, and **active_back_color** specify the colors to be used when the cell changes states, to enhance feedback to the user. The enabled, disabled, and active colors are all foreground colors for the text, and the pen color for the border. The "back" colors are background colors for enabled, disabled and active states. The **back_color** is ignored on 3D cells.

white_space_color specifies the color for various parts of the list where there are no cells, column headings, etc.

rule_color sets the color for horizontal and vertical rules in the list.

If **no_heading** is **TRUE**, then the list will have no column headings.

If **one_row_list** is **TRUE**, then the list will have only one row, regardless of how many form units high the list is, or how many rows would fit on the list. To portably make a one row list without a heading, specify that the list is three * **XI_FU_MULTIPLE** form units high, and set **one_row_list** to **TRUE**. This will make a one row list on every platform. **height** needs to be three * **XI_FU_MULTIPLE** because the smallest one row list on XVT/CH is three * **XI_FU_MULTIPLE** form units high.

If **scroll_bar** is **TRUE**, then the list will have a scroll bar to the right of it, and the application will need to pay attention to the **percent** field when responding to **XIE_GET_FIRST**. The application will also need to respond to **XIE_GET_PERCENT**. See **XIE_GET_FIRST** and **XIE_GET_PERCENT** for further details.

If **sizable_columns** is **TRUE**, then the user can resize columns using the mouse by grabbing onto the border between the column headings.

If **movable_columns** is **TRUE**, then the user can move columns around by grabbing and dropping the column headings.

If **scroll_bar_button** is **TRUE**, then XI will place a small button above the vertical scroll bar, and draw a plus sign in the button. This typically would be used to present the user with a list of available columns that can be added to the list. When the user clicks the scroll bar button, the application will get an **XIE_BUTTON** event, with **xiev->v.xi_obj** set to the list object.

fixed_columns sets the number of columns that will not scroll horizontally. This value is ignored if the list does not scroll horizontally. This typically would be used to fix a row number column and a description column, so that the user always knows which row they are modifying. If the list has movable columns, columns can be moved into and out of the fixed region.

tab_cid is the control ID of the form, list, or container that is next in the meta-tabbing sequence. The focus will go to a control within that object when the character defined in the preference **XI_PREF_ITF_TAB_CHAR** is pressed.

min_cell_height sets the minimum desired cell height, in pixels, so that larger text or icons will fit in the cells without being clipped.

min_heading_height sets the minimum desired height for headings, in pixels, so that larger text or icons will fit in the heading without being clipped.

If **no_horz_lines** is **TRUE**, then XI will not draw the horizontal rules in the list.

If **no_vert_lines** is **TRUE**, then XI will not draw the vertical rules in the list.

start_percent sets the percentage that will be passed to the first **XIE_GET_FIRST** event. In general, it is better to use **xi_scroll_rec** to set initial position in the list. This call can be made during the **XIE_INIT** event in order to avoid extra record events.

first_vis_column sets the column that will appear first in the horizontally scrolling region of the list. This value is ignored if the list does not scroll horizontally.

If **drop_and_delete** is **TRUE**, columns can be deleted from the list by dragging them out of the list region. This value is ignored if the list does not have **movable_columns** set to **TRUE**. Note that if you use this option, you will probably need some mechanism for adding columns (see **scroll_bar_button** above).

If **select_cells** is set to **TRUE**, then users can select a rectangular range of cells. Users can select multiple ranges by holding down the shift key, as they start selections when cells are already selected. When the user selects a range of cells, then XI generates an **XIE_SELECT** event for the list object. The application can query XI for the list of selected cells by calling **xi_get_cell_selection**. Currently, XI only supports selection of visible cells.

If **get_all_records** is set to **TRUE** then XI will generate the **XIE_GET_FIRST** event and all **XIE_GET_NEXT** and **XIE_CELL_REQUEST** events when the list is first initialized. These events will never occur again for that list. This option should only be used for “static” lists since the rows can never be changed once the list is initialized. For example, **xi_delete_row** and **xi_insert_row** can not be called on this kind of list. **NOTE:** the number of rows cannot be greater than 255 and we suggest that the number of rows not exceed 100.

If **resize_with_window** is set to **TRUE**, then XI will resize the list so that the lower right corner of the list will always be in the lower right corner of the client area of the window that contains the list. In this case, the width of the list is ignored, except that it must be set to a non-zero value if you want to enable horizontal scrolling. If the list is not a horizontal scrolling list, then only the vertical size of the list will change as the user sizes the window.

The **font_id** value is used to override the font from the interface. You are responsible for destroying the XVT R4 font, which you can do after the call to xi_create.

horz_sync_list can be set to the control ID of another list in the same interface. In that case, the other list will be scrolled horizontally whenever this list is scrolled horizontally. You will want to set this member in that other list to the control ID of this list. The number and widths of the columns in the two lists must match. If you allow sizable, movable and/or deletable columns, you must respond to the **XIE_COL_SIZE**, **XIE_COL_MOVE**, and/or **XIE_COL_DELETE** events so that the other list can be updated to match.

vert_sync_list can be set to the control ID of another list in the same interface. In that case, the other list will be scrolled vertically whenever this list is scrolled vertically. You will want to set this member in that other list to the control ID of this list. The number and heights of the rows in the two lists must match. If you allow sizable rows, you must respond to the **XIE_ROW_SIZE** event so that the other list can be updated to match. It is nearly impossible to make this option work if **wrap_text** is **TRUE** for any of the columns.

If **row_focus_border** is **TRUE**, the entire row which contains the cell with focus will have a border drawn around it. The color of this border is black unless **row_focus_border_color** is set.

max_lines_in_cell defines the maximum number of lines in cells that have wrapped text in them (see **wrap_text** in **XI_COLUMN_DEF**). This overrides the value for **XI_PREF_DEFAULT_MAX_LINES_IN_CELL**.

If **single_select** is **TRUE**, only one row can be selected at a time. The keyboard can also be used to move the current selection. Up, down, home, end, page up and page down are supported. Every time the selection is changed, an **XIE_SELECT** event will occur for the newly selected row. No "deselect" events will occur for the list. You must still keep track of current selection to respond properly to **XIE_GET_*** events in case the currently selected row is scrolled out of the visible region of the list. The list becomes **READONLY** when this is set.

If **retain_back_color_on_select** is **TRUE**, then when a row is selected, cells with a background color other than the default will retain their color rather than being inverted to indicate selection.

If **draw_and_drop_rows** is **TRUE,** rows can be moved by dragging them and dropping in another position, or another list. They can be deleted by dragging and dropping outside of a list.

If **drag_rows_autoscroll** is **TRUE,** the list will autoscroll as a row is dragged inside of it.

If **button_on_cell_focus** is **TRUE,** the cell button will be drawn only when the **cell** has focus. The default is to draw the cell button when the row containing the cell gains focus.

XI_MAX_EVENT — Maximum Number of XI Events

Summary

```
#define XI_MAX_EVENT 100
```

Description

An application can simulate its own XI events by constructing an **XI_EVENT** struct, and setting the event type to something greater than **XI_MAX_EVENT**. The **user_data** field in the **XI_EVENT** struct can be used to pass data to the event handler.

XI_MOD_* Virtual Key Modifiers

Summary

```
#define XI_MOD_SHIFT    0x10000000
#define XI_MOD_CONTROL  0x20000000
#define XI_MOD_ALT      0x40000000
```

Description

These constants are used in conjunction with the **xi_set_pref** mechanism, to allow the programmer to set preferences to determine which keystrokes invoke certain functions in the interface. These values are bitwise-OR'ed with ASCII or K_* values to produce a composite character description.

Implementation Note

XI_MOD_ALT is not implemented yet. XVT does not indicate whether the ALT key is pressed with character events.

See Also

xi_set_pref, **xi_get_pref**

XI_NULL_OBJ Null Object

Summary

```
#define XI_NULL_OBJ ((XI_OBJ *)NULL)
```

Description

This macro is returned as an error from some XI functions (e.g. **xi_get_obj**) and can be passed as an argument to **xi_create**.

XI_OBJ Object

Summary

```
typedef struct _xi_obj
{
  int cid;
  struct _xi_obj *itf;
  struct _xi_obj *parent;
  XI_OBJ_TYPE type;
  short nbr_children;
  struct _xi_obj * *children;
  long app_data;
  long app_data2;
  union
  {
    XI_CELL_DATA cell;
    XI_ROW_DATA  row_data;
    short        row;
#ifdef XI_INTERNAL
    STRUCT _xi_column_data *column;
    STRUCT _xi_itf_data *itf;
    STRUCT _xi_list_data *list;
    STRUCT _xi_container_data *container;
    STRUCT _xi_btn_data *btn;
    STRUCT _xi_field_data *field;
    STRUCT _xi_form_data *form;
    STRUCT _xi_group_data *group;
    STRUCT _xi_line_data *line;
    STRUCT _xi_rect_data *rect;
    STRUCT _xi_text_data *text;
#else
    /* so that union has room for a ptr */
    char *dummy;
#endif
  } v;
} XI_OBJ;
```

Description

This structure contains an XI object, which can be passed to a variety of functions. You can get a pointer to an object by calling **xi_get_obj** for an interface. Cell and row objects can be initialized using the **XI_MAKE_CELL** and **XI_MAKE_ROW** macros.

WARNING: These values should be treated as read only for any object that comes from the interface. Do not modify the **XI_OBJ** structure unless you are creating it yourself. You should only be creating your own object for cells and rows. Most of these values can be changed using XI function calls. If not, the object can be deleted and recreated.

cid is the control ID of the object.

itf is the interface object that contains the object. This value points to itself if it is the interface object.

parent is the parent of the object. For example, an object of type **XIT_LIST** would be the parent of an object of type **XIT_COLUMN**. The interface object has a **NULL** parent.

type is the type of the object.

nbr_children is the number of objects contained in this object. For example, **nbr_children** would contain the number of columns in a list for an object of type **XIT_LIST**.

children is an array of pointers to the contained objects. For example, **children** would contain an array of pointers to objects of type **XIT_COLUMN** for an object of type **XIT_LIST**.

app_data contains the application data for an object. Call **xi_set_app_data** and **xi_get_app_data** to set or get this data.

app_data2 is similar to **app_data**, but is reserved for internal use.

Any object that was defined for the interface will use one of the **_xi_*_data** pointers to refer to its internal data. These pointers are available only if **XI_INTERNAL** was defined when "xi.h" was compiled. The **type** value determines which of these pointers is valid.

If **XI_INTERNAL** was not defined, **dummy** is used as a placeholder for the pointers that will be defined in the internals of XI. This assures that the union will be the correct size.

If **type** is **XIT_CELL**, then the **cell** member of the union is valid. See **XI_CELL_DATA** for more details. A cell object can be initialized using the **XI_MAKE_CELL** macro.

If **type** is **XIT_ROW**, then the **row** and **row_data** members of the union are valid. See **XI_ROW_DATA** for more details about the **row_data** member. A row object can be initialized using the **XI_MAKE_ROW** macro. Note that the **row** member of the union and the **row** member of the **row_data** structure are actually the same variable. **row** is provided for compatibility with earlier versions of XI. See **XI_ROW_DATA** for more information.

XI_OBJ_DEF Object Definition

Summary

```
typedef struct _xi_obj_def
{
  XI_OBJ_TYPE type;
  int cid;
  struct _xi_obj_def *parent;
  short nbr_children;
  struct _xi_obj_def * *children;
  long app_data;
  long app_data2;
  union
  {
    XI_BTN_DEF *btn;
    XI_CONTAINER_DEF *container;
    XI_FORM_DEF *form;
    XI_FIELD_DEF *field;
    XI_GROUP_DEF *group;
    XI_LINE_DEF *line;
    XI_RECT_DEF *rect;
    XI_TEXT_DEF *text;
    XI_COLUMN_DEF *column;
    XI_ITF_DEF *itf;
    XI_LIST_DEF *list;
  } v;
} XI_OBJ_DEF;
```

Description

This structure is used for the definition of any object in XI. Normally, it is created by one of the convenience functions (**xi_add_*_def** or **xi_create_itf_def**).

type contains the object type. This also determines which of the pointers within the **v** union is valid.

cid contains the control ID of the object.

parent contains a pointer to the parent object definition. If the convenience function was called with a parent of **NULL**, then **parent** will also be **NULL**.

nbr_children and **children** fields specify a list of child XI objects. These values are updated by the convenience functions whenever an object is specified as the parent. If you want to update these values, you should use tree memory functions (see **xi_tree_***) for the **children** array and be sure that the memory is parented to the definition.

app_data is an arbitrary piece of data that the application can set for the interface. This data can be retrieved from the created object by calling **xi_get_app_data**.

app_data2 is similar to **app_data**, but is reserved for internal use.

btn points to the button definition, but only if the **type** is **XIT_BTN**.

container points to the button container definition, but only if the **type** is **XIT_CONTAINER**.

form points to the form definition, but only if the **type** is **XIT_FORM**.

field points to the edit field definition, but only if the **type** is **XIT_FIELD**.

group points to the group definition, but only if the **type** is **XIT_GROUP**.

line points to the line definition, but only if the **type** is **XIT_LINE**.

rect points to the rectangle definition, but only if the **type** is **XIT_RECT**.

text points to the static text definition, but only if the **type** is **XIT_TEXT**.

column points to the column definition, but only if the **type** is **XIT_COLUMN**.

itf points to the interface definition, but only if the **type** is **XIT_ITF**.

list points to the list definition, but only if the **type** is **XIT_LIST**.

See Also

XI_OBJ_TYPE, **XI_*_DEF**

XI_OBJ_TYPE Object Type

Summary

```
typedef enum _xi_obj_type
{
  XIT_BTN,
  XIT_CONTAINER,
  XIT_FORM,
  XIT_FIELD,
  XIT_GROUP,
  XIT_LINE,
  XIT_RECT,
  XIT_TEXT,
  XIT_CELL,
  XIT_COLUMN,
  XIT_ITF,
  XIT_LIST,
  XIT_ROW
} XI_OBJ_TYPE;
```

Description

XI_OBJ_TYPE is an enumeration of all the types of objects in XI. This type appears in the **XI_OBJ_DEF** and **XI_OBJ** structures.

XIT_BTN: Refers to a variety of button controls. The specific kind of button will be defined by **XI_BTN_TYPE**. Buttons are defined with the **XI_BTN_DEF** structure.

XIT_CONTAINER: Refers to a button container. Button containers are defined with the **XI_CONTAINER_DEF** structure.

XIT_FORM: Refers to a form, which is used to hold edit fields. Forms are defined by the **XI_FORM_DEF** structure.

XIT_FIELD: Refers to an edit field. Edit fields are defined by the **XI_FIELD_DEF** structure.

XIT_GROUP: Refers to a group control, which is used to generate on and off focus events for a group of other controls. Groups are defined by the **XI_GROUP_DEF** structure.

XIT_LINE: Refers to a static line control. Lines are defined by the **XI_LINE_DEF** structure.

XIT_RECT: Refers to a static rectangle control. Rectangles are defined by the **XI_RECT_DEF** structure.

XIT_TEXT: Refers to a static text control. Text is defined by the **XI_TEXT_DEF** structure.

XIT_CELL: Refers to a cell. Cell objects are not "real" objects since they actually refer to a row within a list column. Since cell objects refer to a particular position that is currently visible, they are invalid as soon as the list scrolls. There is no such thing as a cell definition. Cell objects are defined by the **XI_CELL_DATA** structure.

XIT_COLUMN: Refers to a column in a list. Columns are defined by the **XI_COLUMN_DEF** structure.

XIT_ITF: Refers to an interface object. The interface is essentially the same as the window itself and it contains all other controls as either direct or indirect children. The interface is defined by the **XI_ITF_DEF** structure.

XIT_LIST: Refers to a list control. Lists can contain column controls. Lists are defined by the **XI_LIST_DEF** structure.

XIT_ROW: Refers to a row. Row objects are not "real" objects since they actually refer to a row within a list. Since rows refer to a particular position that is currently visible, they are invalid as soon as the list scrolls. There is no such thing as a row definition. Row objects are defined by the **row** member of the **XI_OBJ** structure.

XI_PNT XI Point (Form Units)

Summary

```
typedef PNT XI_PNT;
```

Description

This type is used to hold coordinates of XI objects. All XI objects are defined in terms of form units, and using the type **XI_PNT** implies these coordinates.

Structures of type **XI_PNT** are defined to have the same fields as the XVT data type **PNT**. See your XVT documentation for the structure of **PNT**.

XI_RCT XI Rectangle (Form Units)

Summary

```
typedef XinRect XI_RCT;
```

Description

This type is used to hold coordinates of XI objects. All XI objects are defined in terms of form units, and using the type **XI_RCT** implies these coordinates.

Structures of type **XI_RCT** are defined to have the same fields as the XVT datatype **RCT**. See your XVT documentation for the structure of **RCT**.

XI_PREF_TYPE Preference Type

Summary

```
typedef enum
{
  XI_PREF_OVERLAP,
  XI_PREF_FORM_TAB_CHAR,
  XI_PREF_FORM_BACKTAB_CHAR,
  XI_PREF_SCROLL_INC,
  XI_PREF_3D_LOOK,
  XI_PREF_USE_APP_DATA,
  XI_PREF_AUTOSEL_ON_MOUSE,
  XI_PREF_CELL_BTN_ICON_X,
  XI_PREF_CELL_BTN_ICON_Y,
#ifdef XI_SUPPORT_R3_API
  XI_PREF_R4_API,
#endif
  XI_PREF_SINGLE_CLICK_COL_SELECT,
  XI_PREF_COLUMN_OFFSET,
  XI_PREF_SB_OFFSET,
  XI_PREF_SB_WIDTH,
  XI_PREF_SB_HEIGHT,
  XI_PREF_SIZE_CURSOR_RID,
  XI_PREF_HAND_CURSOR_RID,
  XI_PREF_VSIZE_CURSOR_RID,
  XI_PREF_COMBO_ICON,
  XI_PREF_COLOR_LIGHT,
  XI_PREF_COLOR_CTRL,
  XI_PREF_COLOR_DARK,
  XI_PREF_OPTIMIZE_CELL_REQUESTS,
  XI_PREF_CARET_WIDTH,
```

```
  XI_PREF_TRIPLE_CLICK_TIME,
  XI_PREF_BUTTON_KEY,
  XI_PREF_LIMIT_MIN_WIN_SIZE,
  XI_PREF_DEFAULT_MAX_LINES_IN_CELL,
  XI_PREF_KEEP_FOCUS_FIXED,
  XI_PREF_NATIVE_CTRLS,
  XI_PREF_ITF_TAB_CHAR,
  XI_PREF_ITF_BACKTAB_CHAR,
  XI_PREF_ITF_WS_RIGHT,
  XI_PREF_ITF_WS_BOTTOM,
  XI_PREF_VIR_SP_H,
  XI_PREF_VIR_SP_V,
  XI_PREF_DBL_PRESSES_BUTTON,
  XI_PREF_CONTAINER_GRID_WIDTH,
  XI_PREF_MULTILINE_QUICK_PASTE,
  XI_PREF_COLOR_DISABLED,
  XI_PREF_BUTTON_HEIGHT,
  XI_PREF_BUTTON_PAD,
  XI_PREF_HORZ_SPACING,
  XI_PREF_VERT_SPACING,
  XI_PREF_HORZ_PIXEL_SPACING,
  XI_PREF_VERT_PIXEL_SPACING,
  XI_PREF_ITF_MIN_TOP,
  XI_PREF_ITF_MIN_LEFT,
  XI_PREF_XIL,
  XI_PREF_ASSERT_ON_NULL_CID,
  XI_PREF_NO_BUTTON_SPACE,
  XI_PREF_TAB_USE_UNCHECKED_COLORS,
  XI_PREF_TAB_UNCHECKED_COLOR_LIGHT,
  XI_PREF_TAB_UNCHECKED_COLOR_CTRL,
  XI_PREF_TAB_UNCHECKED_COLOR_DARK,
  XI_PREF_TAB_CHANGE_CHECKED_FONT,
  XI_PREF_TAB_WIN95_LOOK,
  XI_PREF_TAB_MOTIF_LOOK,
  XI_PREF_TAB_WIN31_LOOK,
  XI_PREF_USE_ALTERNATING_ROWS,
  XI_PREF_ALTERNATE_ROW_COLOR,
  XI_PREF_LASTPREF          /* should always be last */
} XI_PREF_TYPE;
```

Description

Members of this enumeration are passed to the functions **xi_get_pref** and **xi_set_pref** to allow application control over some aspects of XI. These settings affect all of XI and some must be set before the call to **xi_init**. The default value is shown in parenthesis after each preference. Some default values vary from platform to platform, so they will be indicated as (*).

XI_PREF_OVERLAP (1): The number of lines in a list that will overlap during page-up and page-down operations.

XI_PREF_FORM_TAB_CHAR (\t): Defines the character that moves the focus forward from controls that can have focus (e.g. edit fields and buttons) to the next control that can have focus.

XI_PREF_FORM_BACKTAB_CHAR (K_BTAB): Defines the character that moves the focus backwards from controls that can have focus (e.g. edit fields and buttons) to the previous control that can have focus.

XI_PREF_SCROLL_INC (1): Defines the number of lines the list scrolls when the user moves off of the top or bottom of the list using the arrow keys.

XI_PREF_3D_LOOK (TRUE): This preference determines whether rectangles and lines are drawn with the 3D look. This value affects only rectangles and lines. If you wish to have the 3D look for radio buttons and check boxes, set **XI_PREF_NATIVE_CTRLS** to **FALSE**.

XI_PREF_USE_APP_DATA (FALSE): This preference determines whether XI will put the interface object in the application data for a window. If you wish to use the app data for all XVT windows, including windows that contain an XI interface, set this preference to **FALSE** before calling **xi_init**.

XI_PREF_AUTOSEL_ON_MOUSE (FALSE): If this preference is set to **TRUE**, then when the user clicks on an autoselect edit field, first the entire edit field is selected. If the user clicks again on the edit field, then the insertion point is set. This works the same way for cells in a list.

XI_PREF_CELL_BTN_ICON_X (1): Defines an offset for the icon in cell buttons. If the appearance of the icon is improved by moving it toward the right, within the cell button, then set this preference.

XI_PREF_CELL_BTN_ICON_Y (1): Defines an offset for the icon in cell buttons. If the appearance of the icon is improved by moving it toward the bottom, within the cell button, then set this preference.

XI_PREF_R4_API (*): This preference is set automatically based on compile time defines. It should not be set, by the user, at run-time.

XI_PREF_SINGLE_CLICK_COL_SELECT: This preference determines if a single mouse down will result in a **XIE_SELECT** event when a column is movable. Normally a movable column requires a double mouse down to produce a XIE_SELECT event.

XI_PREF_COLUMN_OFFSET (*): This preference defines the number of pixels between the column rules and the cell text in a list.

XI_PREF_SB_OFFSET: This preference is essentially obsolete and should be ignored.

XI_PREF_SB_WIDTH: This preference is essentially obsolete and should be ignored.

XI_PREF_SB_HEIGHT: This preference is essentially obsolete and should be ignored.

XI_PREF_SIZE_CURSOR_RID (XI_CURSOR_RESIZE): Defines the resource ID for the horizontal size cursor, used when resizing a column. If you wish to design your own cursor resource, and give it a different resource ID than the default, set this preference with the resource ID of your cursor. If set to zero, the cursor will not be changed. On platforms without cursors, the default value is zero.

XI_PREF_HAND_CURSOR_RID (XI_CURSOR_HAND): Defines the resource ID for the hand cursor, used when moving columns. If you wish to design your own cursor resource, and give it a different resource ID than the default, set this preference with the resource ID of your cursor. If set to zero, the cursor will not be changed. On platforms without cursors, the default value is zero.

XI_PREF_VSIZE_CURSOR_RID (XI_CURSOR_VRESIZE): Defines the resource ID for the vertical size cursor, used when resizing a row. If you wish to design your own cursor resource, and give it a different resource ID than the default, set this preference with the resource ID of your cursor. If set to zero, the cursor will not be changed. On platforms without cursors, the default value is zero.

XI_PREF_COMBO_ICON (COMBO_ICON): Defines the resource ID for a combo button for an edit field or for a cell. If you wish to design your own icon for edit field buttons or cell buttons, then set this preference to the resource ID of your icon. On platforms without icons, the default value is zero.

XI_PREF_COLOR_LIGHT (*): This preference defines the color used for the highlight part of 3D lines, rectangles and diamonds(checkboxes). This preference may be overridden for **XIT_BTN**, **XIT_FIELD** and **XIT_RECT** objects. Usually this defaults to **COLOR_WHITE**.

XI_PREF_COLOR_CTRL (*): This preference defines the color used for the flat part of 3D rectangles. This preference may be overridden for **XIT_BTN**, **XIT_FIELD** and **XIT_RECT** objects. The default is usually a shade of gray.

XI_PREF_COLOR_DARK (*): This preference defines the color used for the shadow part of 3D lines, rectangles and diamonds(checkboxes). This preference may be overridden for **XIT_BTN**, **XIT_FIELD** and **XIT_RECT** objects. The default is usually a shade of gray that is darker than the gray used for **XI_PREF_COLOR_CTRL**.

XI_PREF_OPTIMIZE_CELL_REQUESTS (TRUE): If set to **TRUE**, cell request events will occur only when a cell needs to be drawn. Cells that are not currently visible will not have cell request events until they appear. If set to **FALSE**, cell request events will occur for all cells in a visible row.

XI_PREF_CARET_WIDTH (1): Defines the width of the caret within XI. It is passed to **xvt_win_set_caret_size**.

XI_PREF_TRIPLE_CLICK_TIME (500): Defines the amount of time, in milliseconds, that will be allowed between a double click event and the next mouse down event in order to register as a triple click. A triple click will select all text in a cell or edit field. If this value is set to zero, triple click detection will be disabled.

XI_PREF_BUTTON_KEY (*): Defines the keyboard character that will “press” a cell button or edit field button.

XI_PREF_LIMIT_MIN_WIN_SIZE (TRUE): If set to **TRUE**, the minimum size of the window will be limited based on a list inside that window. This option is only active if the **resize_with_window** option was used for the list. In that case, the window size will be limited so that the entire list can be seen.

XI_PREF_DEFAULT_MAX_LINES_IN_CELL (5): Sets the maximum number of lines in a cell for wrapped text. This value can be overridden for a list by setting the **max_lines_in_cell** value in the **XI_LIST_DEF** structure. If, at creation time, the list does not have columns with wrapped text then the default is 1.

XI_PREF_KEEP_FOCUS_FIXED (FALSE): If set to **TRUE**, the cell with focus on a list will keep focus when the list is scrolled, even if the cell is no longer visible. Typing and keyboard navigation will scroll the list back to that cell, if necessary. If you use this option, be sure to set the **has_focus** member of the event structure for **XIE_GET_*** events.

XI_PREF_NATIVE_CTRLS (TRUE): This preference determines whether XI draws its own buttons, radio buttons, and check boxes, or if XI uses native controls.

XI_PREF_ITF_TAB_CHAR (K_F3): This preference defines the meta-tab character, which moves the focus forward from a list, form, or container to a control within the next list, form, or container. In this way, the user can use the keyboard to move from one higher level object to another.

XI_PREF_ITF_BACKTAB_CHAR (K_F4): This preference defines the back meta-tab character, which moves the focus backwards from a list, form, or container to a control within the previous list, form, or container. In this way, the user can use the keyboard to move from one higher level object to another.

XI_PREF_ITF_WS_RIGHT (16): This preference defines the amount of space, in form units, that XI will place to the right of the rightmost object when XI sizes a window in **xi_create**. This value can be overridden for a particular interface by setting the **whitespace_right** member of the interface definition.

XI_PREF_ITF_WS_BOTTOM (4): This preference defines the amount of space, in form units, that XI will place below the bottommost object when XI sizes a window in **xi_create**. This value can be overridden for a particular interface by setting the **whitespace_bottom** member of the interface definition.

XI_PREF_VIR_SP_H (40): When navigating in a virtual interface, XI always attempts to make the object with the focus be visible. This preference defines how much space, in pixels, to make visible to the right and left of the object with the focus.

XI_PREF_VIR_SP_V (20): When navigating in a virtual interface, XI always attempts to make the object with the focus be visible. This preference defines how much space, in pixels, to make visible above and below the object with the focus.

XI_PREF_DBL_PRESSES_BUTTON (TRUE): This preference determines if a double click with the mouse will press a button, radio button, or check box. This preference is ignored if **XI_PREF_NATIVE_CTRLS** is set to **TRUE**.

XI_PREF_CONTAINER_GRID_WIDTH (1): This preference defines the rule width between buttons in a grid container.

XI_PREF_MULTILINE_QUICK_PASTE (TRUE): If set to **TRUE**, paste commands will be passed directly to multiline edit fields. If set to **FALSE**, the characters will be added one at a time after each successful **XIE_CHAR_FIELD** event.

XI_PREF_COLOR_DISABLED (COLOR_GRAY): This color is used as the foreground color for text on disabled **XIT_TEXT** and **XIT_BTN** controls.

XI_PREF_BUTTON_HEIGHT (*): This preference defines the height of buttons, in pixels, that are stacked vertically in a container.

XI_PREF_BUTTON_PAD (*): When placing buttons arranged horizontally in a container, XI determines the maximum length of the text in all of the buttons, and adds this preference to that maximum length. All of the buttons in the container are made as wide as this length. In other words, this preference defines the amount of white space to the right and left of the text of the widest button.

XI_PREF_HORZ_SPACING (*): This preference defines the number of form units between horizontal buttons in a container. This value is only used if **XI_PREF_HORZ_PIXEL_SPACING** is set to zero.

XI_PREF_VERT_SPACING (*): This preference defines the number of form units between vertical buttons in a container. This value is only used if **XI_PREF_VERT_PIXEL_SPACING** is set to zero.

XI_PREF_HORZ_PIXEL_SPACING (0): This preference defines the number of pixels between horizontal buttons in a container. If set to zero, **XI_PREF_HORZ_SPACING** is used.

XI_PREF_VERT_PIXEL_SPACING (0): This preference defines the number of pixels between vertical buttons in a container. If set to zero, **XI_PREF_VERT_SPACING** is used.

XI_PREF_ITF_MIN_TOP (*): This preference defines the minimum number of pixels above a window when XI places a window in **xi_create**.

XI_PREF_ITF_MIN_LEFT (*): This preference defines the minimum number of pixels to the left of a window when XI places a window in **xi_create**.

XI_PREF_XIL (FALSE): Changes the behavior of the list control to match that of the XVT PowerObjects version of the XI spreadsheet. This value can be overridden for particular interface by setting the **use_xil_win** member of the interface definition.

XI_PREF_ASSERT_ON_NULL_CID (FALSE): Causes xi_get_obj to assert if an invalid cid is passed in.

XI_PREF_NO_BUTTON_SPACE (FALSE): When this preferenc is set, a field button is drawn inside the field, rather on an outside edge. When using this preference, you must adjust the size of the field to accommodate the button.

XI_PREF_TAB_USE_UNCHECKED_COLORS (FALSE): This preference determines which colors are used for unchecked buttons of type **XIBT_TABBTN**. If it is **FALSE**, **XI_PREF_COLOR_*** preferences are used. If it is **TRUE,** **XI_PREF_TAB_UNCHECKED_COLOR_*** preferences are used.

XI_PREF_TAB_UNCHECKED_COLOR_ LIGHT (*): This preference defines the color used for the highlight part for buttons of type **XIBT_TABBTN**. It applies only to buttons that are not checked. This color is used only if **XI_PREF_TAB_USE_UNCHECKED_COLORS** is **TRUE**. The default is **XI_PREF_COLOR_LIGHT**.

XI_PREF_TAB_UNCHECKED_COLOR_CTRL (*): This preference defines the color used for the flat part for buttons of type **XIBT_TABBTN**. It applies only to buttons that are not checked. This color is used only if **XI_PREF_TAB_USE_UNCHECKED_COLORS** is **TRUE**. The default is **XI_PREF_COLOR_CTRL**.

XI_PREF_TAB_UNCHECKED_COLOR_DARK (*): This preference defines the color used for the shadow part of buttons of type **XIBT_TABBTN**. It applies only to buttons that are not checked. This color is used only if **XI_PREF_TAB_USE_UNCHECKED_COLORS** is **TRUE**. The default is **XI_PREF_COLOR_DARK**.

XI_PREF_TAB_CHANGE_CHECKED_FONT (TRUE): This preference determines if the text of a checked button of type **XIBT_TABBTN** is drawn in a bold font. If it is **TRUE**, the font is bold.

XI_PREF_TAB_WIN95_LOOK (TRUE): This preference determines how the outline for buttons of type **XIBT_TABBTN** is drawn. If it is **TRUE**, the buttons are drawn with a rectangular look. This is the default look for **XIBT_TABBTN**.

XI_PREF_TAB_MOTIF_LOOK (TRUE): If this preference is **TRUE**, the outline for buttons of type **XIBT_TABBTN** are drawn with a rectangular look. The buttons have a sloped edge on both sides.

XI_PREF_TAB_WIN31_LOOK (TRUE): If this preference is **TRUE**, the outline for buttons of type **XIBT_TABBTN** are drawn with a beveled look. The button sides have a pronounced slope.

XI_PREF_USE_ALTERNATING_ROWS (TRUE): If this preference is **TRUE**, rows are drawn using alternating colors.

XI_PREF_ALTERNATE_ROW_COLOR (0x00F0FCFCL): If XI_PREF_USE_ALTERNATING_ROWS is **TRUE**, this is the color of every second row.

XI_RECT_DEF Rect Object Definition

Summary

```
typedef struct _xi_rect_def
{
  XinRect xi_rct;
  XinRect pixel_rect;
  XinColor fore_color;
  XinColor back_color;
  XinColor hilight_color;
  XinColor shadow_color;
  BOOLEAN well;
  BOOLEAN ridge;
  XI_BITMAP* bitmap;
  unsigned long attrib;
} XI_RECT_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. Use this structure if you are modifying an existing rectangle definition. Otherwise, use the convenience function **xi_add_rect_def**, which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

xi_rct is the bounding rectangle, in form units. This rectangle is ignored if **pixel_rect** is specified.

pixel_rect is the bounding rectangle, in pixels.

If the 3D appearance is not used, then **fore_color** is the foreground color of the rectangle, and **back_color** is the background color of the rectangle.

If the 3D appearance is used, then by default, the colors of the rectangle are determined by the preferences **XI_PREF_COLOR_LIGHT**, **XI_PREF_COLOR_CTRL**, and **XI_PREF_COLOR_DARK**. However, if you set **hilight_color**, **back_color**, and **shadow_color**, the preferences are overridden, and you can create rectangles that each have their own 3D coloring.

If the 3D appearance is used, then rectangles either appear as “wells”, “platforms” or “ridges”. If **well** is set to **TRUE**, then the line has the appearance of being indented. If **well** is not set, then the line has the appearance of coming out of the screen. If **ridge** is set to **TRUE**, the lines around the rectangle have a “ridge” appearance.

bitmap is a pointer to the bitmap created with **xi_bitmap_create** that will be drawn in the rectangle. The local copy of the bitmap is destroyed by calling **xi_def_free**, after the call to **xi_create**.

attrib is an OR’ed combination of **XI_ATR_*** values which control the behavior of the rectangle. The following attributes apply to rectangles:

```
XI_ATR_VISIBLE
```

XI_ROW_DATA Row Data Specification

Summary

```
typedef struct _xi_row_data
{
  short         row;
  unsigned char is_vert_scrolled;
} XI_ROW_DATA;
```

Description

This is used to specify a row object. (Note: This structure supercedes the older **row** value in the **XI_OBJ** structure, but both are supported.)

row is the row number of the row. This row number refers to its position in the visible part of the list. If you want the record handle for that row, use **xi_get_list_info** to get the record handle array and use **row** to index that array.

is_vert_scrolled will be **TRUE** if the row is not currently visible. This will only happen if you are using the **XI_PREF_KEEP_FOCUS_FIXED** option and this row has the cell with focus and it is scrolled off the list vertically.

See Also

XI_MAKE_ROW, **XI_OBJ**

XI_SCROLL_* Methods to Scroll a List

Summary

```
#define XI_SCROLL_PGUP    1001
#define XI_SCROLL_PGDOWN  1002
#define XI_SCROLL_FIRST   1003
#define XI_SCROLL_LAST    1004
```

Description

These constants are passed to **xi_scroll** to indicate that special actions should be taken such as scrolling in pages, or scrolling to the top or bottom of the list.

See Also

xi_scroll

XI_TEXT_DEF Text Object Definition

Summary

```
typedef struct _xi_text_def
{
  XinRect    xi_rct;
  XinRect    pixel_rect;
  unsigned long attrib;
  char*    text;
  XinColor fore_color;
  XinColor back_color;
  char     mnemonic;
  short    mnemonic_instance;
  XVT_FNTID font_id;
} XI_TEXT_DEF;
```

Description

This structure is part of the union within the **XI_OBJ_DEF** structure. Use this structure if you are modifying an existing static text definition. Otherwise, use the convenience function **xi_add_text_def**, which automatically allocates this structure along with the associated **XI_OBJ_DEF**.

xi_rct defines the position and bounding rectangle of the text object, in form units. This rectangle is ignored if **pixel_rect** is defined.

pixel_rect defines the position and bounding rectangle of the text object, in pixels.

attrib is an OR'ed combination of **XI_ATR_*** values which control the appearance of the text. The following attributes apply to static text controls:

```
XI_ATR_ENABLED
XI_ATR_RJUST
XI_ATR_VISIBLE
```

text is the displayed text.

fore_color is the color used for the text object.

back_color is the background color for the text object.

mnemonic defines the character, used in combination with the ALT key, that will activate a field.

mnemonic_instance defines which instance of the mnemonic character to underline. Setting this to 1 will cause the first instance of the mnemonic to be underlined.

The **font_id** value is used to override the font from the interface. You are responsible for destroying the XVT R4 font, which you can do after the call to xi_create.

XI_VERSION Version String

Summary

```
#define XI_VERSION "4.0"
```

Description

This defines the version number of XI as a string constant.

XI_VERSION_NBR Version Number

Summary

```
#define XI_VERSION_NBR 4.0
```

Description

This defines the version number of XI as a floating point constant.

XinBitmap XI Bitmap

Summary

```
Internal structure
```

Description

Type **XinBitmap** is defined internal to XI. It is the data type which is used with xi_bitmap_* functions.

XinBorderStyle Style of window to create

Summary

```
typedef enum
{
  XinBorderDouble,
```

```
  XinBorderSingle,
  XinBorderSizable,
  XinBorderFixed,
  XinBorderNone         /* NOTIMP -- requires child window support
*/
} XinBorderStyle;
```

Description

This the type of interface window to create.

XinBorderSizable: Is a document window, which has a size control.

XinBorderSingle: Is a plain window that does not have a close box.

XinBorderDouble: Is a double border window that does not have a close box.

XinBorderNone: Is a window that does not have a border or close box.

XinBorderFixed: Is a document window.

XinColor XI Color

Summary

```
typedef unsigned long XinColor;
```

Description

Type **XinColor** is defined to have the same type as the XVT datatype **COLOR**.

XinPoint XI Point (Form Units)

Summary

```
typedef XI_PNT XinPoint;
```

Description

Type **XinPoint** is defined to have the same type as the XI datatype **XI_PNT**.

XinRect — XI Rectangle (Form Units)

Summary

```
typedef XI_RCT XinRect;
```

Description

Type **XinRect** is defined to have the same type as the XI datatype **XI_RCT**.

XinWindow — XI Window

Summary

```
define XinWindow WINDOW
```

Description

Type **XinWindow** is defined to have the same type as the XVT datatype **WINDOW**.

Chapter 2

XI Events

XI events are passed to the event handler function that is associated with the interface. The **XI_EVENT** structure contains the **type** of the event. Each section in this chapter is one of the possible event types. XI provides information about the event in the **v** union. Some values may also be set in the event structure which are used by XI to continue processing after the event. The *Description* details the parts of the event structure that are used for each event.

Some events can be **refused**. You can refuse an event by setting the **refused** value in the **XI_EVENT** structure to **TRUE**. Each event has a *Refusable* section that describes the effects of refusing that event.

XIE_BUTTON Button Pressed

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event indicates that a button was pressed.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is an **XIT_BTN** if the button pressed was a pushbutton, radio button, check box, or tab button.

xi_obj is an **XIT_FIELD** if the button pressed was an edit field button.

xi_obj is an **XIT_LIST** if the button pressed was the scroll bar button that is above the vertical scroll bar of a list.

xi_obj is an **XIT_CELL** if the button pressed was a cell button.

Refusable

This event cannot be refused. However, if you do nothing in this event, the button press will have not effect.

Example

The following code is from “lstlink.c”.

```
void form_process_button( XI_OBJ* itf, XI_OBJ* button )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( button->cid )
  {
    case SAVE_BTN_CID:
      if ( !xi_move_focus( itf ) )
        break;
      if ( form_info->changed )
        update_record( xi_get_obj( itf, FORM_CID ),
                       form_info->handle );
      refresh_row( form_info->parent_list, form_info->handle );
      form_info->changed = FALSE;
      xi_delete( itf );
      break;
    case CANCEL_BTN_CID:
      if ( form_info->changed && xvt_ask( "No", "Yes", NULL,
         "You have made changes. Are you sure you want to cancel?" )
         != RESP_2 )
        break;
      xi_delete( itf );
      break;
    case SECTION_ONE_CID:
    case SECTION_TWO_CID:
    {
      SECTION_INFO* info = (SECTION_INFO*)xi_get_app_data( button );

      if ( form_info->cur_section != info )
      {
        change_section( form_info->cur_section, FALSE );
        change_section( info, TRUE );
        form_info->cur_section = info;
      }
      xi_check( button, TRUE );
      break;
    }
```

```
    case FIELD_BASE_CID + LINK_WHO:
      create_who_list( button, form_eh );
      break;
  }
}

static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_BUTTON:
      form_process_button( itf, xiev->v.xi_obj );
      break;
    ...
  }
}
```

XIE_CELL_REQUEST Request for Cell Information

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_cell_request
    {
      struct _xi_obj *list;
      char *s;
      short len;
      long rec;
      short col_nbr;
      int icon_rid;
      XI_BITMAP* bitmap;
      unsigned long attrib;
      XinColor color;
      XinColor back_color;
      BOOLEAN button;
      BOOLEAN button_on_left;
      BOOLEAN button_on_focus;
      BOOLEAN button_full_cell;
      int button_icon_rid;
      XI_BITMAP* button_bitmap;
```

```
      long *records;
      XVT_FNTID font_id;
} cell_request;
    ...
  } v;
} XI_EVENT;
```

Description

The purpose of this event is to request text or information for each of the individual cells of a row after the application has fulfilled one of the record request events (**XIE_GET_FIRST**, **XIE_GET_LAST**, **XIE_GET_NEXT**, or **XIE_GET_PREV**). In other words, the application does not give XI the text for the cells at the time of the record request, but instead, XI first gets the records, then requests the text or information for each of the cells.

This event uses the **cell_request** member of the union **v** in **XI_EVENT**.

list is the list object that contains the requested cell.

s should be set by your program to contain the text for the cell. If the cell is not variable length, **s** is already pointing at an appropriately sized buffer. The buffer is **len** characters, including the '\0' which must be included in the text. If **var_len_text** was set for the column, which contains the cell, you should resize **s** to handle the string you are passing in. You must use **xi_tree_malloc** with a NULL parent to do this, for example:

```
len = strlen (db[xiev->v.cell_request.rec].name);
if (len != xiev->v.cell_request.len)
  xiev->v.cell_request.s =
       (char*)xi_tree_malloc( (len + 1), NULL);   }
strncpy(xiev->v.cell_request.s,db[xiev->v.cell_request.rec].name,
   len);
```

rec is the record handle that was returned by one of the record request events.

col_nbr is the column number. Since columns may be movable, you should retrieve the column ID by accessing the **XI_OBJ** structure for the column, for example:

```
switch ( xiev->v.cell_request.list->children[
         xiev->v.cell_request.col_nbr ]->cid )
{
  ...
```

icon_rid should be set if you want an icon to appear in that cell, instead of text. Or, you can use a **bitmap.** The bitmap is created by call xi_create_bitmap. You are responsible for destroying the bitmap pointer. The three techniques listed below, for fonts will also work for bitmaps.

attrib is an OR'ed combination of **XI_ATR_*** values which control some of the behavior of the cell.The following attributes apply to cells:

```
XI_ATR_RJUST
XI_ATR_SELECTED
XI_ATR_HCENTER
```

color and **back_color** are the foreground and background colors for text in the cell. On platforms that do not have full color icons, these colors will also set the foreground and background colors of the icon.

The **font_id** value is used to override the default font for the list on a particular cell. The **font_id** member is a XVT R4 font. You are responsible for destroying the XVT R4 font, but you cannot do so until after XI has finished processing the results of the cell request. There are three techniques for accomplishing this:

1. Create fonts globally when the application starts and destroy them on termination.

2. Keep track of the created fonts for an interface in its application data and destroy them during the **XIE_CLEANUP** event.

3. Keep track of the created fonts for a row in the list and store the information in a structure that the record handle points to. Destroy the fonts in the **XIE_REC_FREE** event.

If **button** is **TRUE**, the cell will have a cell button. This button will appear on the right side of the cell, unless **button_on_left** is set to **TRUE**. If **button_full_cell** is set to **TRUE** the button will fill the cell. If **button_on_focus** is set to **TRUE**, the button will only appear when this cell or some other cell in the same row has focus. When the user presses the cell button, an **XIE_BUTTON** event is generated for the cell. By default, the cell button uses the **XI_PREF_COMBO_ICON** value for the icon in the cell button. If **button_icon_rid** is set, that icon will be used instead. If **button_bitmap** is set, the bitmap will be used instead of a icon.

The array of record handles is returned in **records**. This is the same array that is returned by the function xi_get_list_info.

Refusable

This event cannot be refused. Records should be refused in the record request events (**XIE_GET_FIRST**, **XIE_GET_LAST**, **XIE_GET_NEXT**, and **XIE_GET_PREV**), not in this event.

Example

The following code is from “lstcol.c”.

```
static void do_cell_request( COLUMN_LIST_INFO* list_info,
                             XI_EVENT* event )
{
  struct xit_cell_request* request = &event->v.cell_request;
  int                      count;
  XI_OBJ**                 members;
  int                      column_id;
  COLUMN_INFO*             column_info;

  members = xi_get_member_list( request->list, &count );
  column_id = members[ request->col_nbr ]->cid;
  column_info = list_info->columns + (int)(request->rec);
  switch (column_id)
  {
    case SELECT_COL_ID:
      if (column_info->selected)
        request->icon_rid = ICON_CHECK;
      else
        request->icon_rid = ICON_EMPTY;
      break;
    case NAME_COL_ID:
      strncpy( request->s, column_info->column_definition
               ->v.column->heading_text, request->len - 1 );
      break;
  }
}

static void list_eh( XI_OBJ* itf, XI_EVENT* event )
{
  COLUMN_LIST_INFO* list_info =
                   (COLUMN_LIST_INFO*)xi_get_app_data( itf );

  switch ( event->type )
  {
    ...
    case XIE_CELL_REQUEST:
      do_cell_request( list_info, event );
      break;
    ...
  }
}
```

XIE_CHAR_CELL Character for Cell

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_chr
    {
      struct _xi_obj *xi_obj;
      int ch;
      BOOLEAN shift;
      BOOLEAN control;
      BOOLEAN is_paste;
    } chr;
    ...
  } v;
} XI_EVENT;
```

Description

This event occurs whenever an cell has focus and a character is typed for that cell or a paste operation is done on that cell. This event allows you to modify and/or validate the character before it is inserted into the current text for the cell. Movement characters, such as left and right arrows, home and end, also generate character events. If the event is refused for these movement characters, the movement will not occur.

This event uses the **chr** member of the union **v** in **XI_EVENT**.

xi_obj is the cell object that has focus and is receiving a character.

ch is the character. This value may be modified. If so, that new character will be put into the cell text instead of the original one. For example, you may wish to force lower case to upper case.

shift is **TRUE** if the shift key was being held when the character was typed.

control is **TRUE** if the control key was being held when the character was typed.

is_paste is **TRUE** if the character is the result of a paste command, rather than an actual character typed on the keyboard. If this value is **TRUE**, you may want to avoid beeps, dialogs, etc. because there will usually be many character events in a row during a paste operation.

Refusable

This event can be refused. If it is refused, the character will not be inserted into the cell text or, if the character would move the insertion point, that movement does not occur.

Example

The following code is from “lstlink.c”.

```
static BOOLEAN validate_date_char( int ch )
{
  if ( ch > 255 || !isprint( ch ) )
    return TRUE;
  if ( ch == '/' || ch == '-' || isdigit( ch ) )
    return TRUE;
  return FALSE;
}

static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_CHAR_CELL:
    {
      XI_OBJ*    cell = xiev->v.chr.xi_obj;
      LINK_FIELD field = column_to_field( cell->parent,
                         cell->v.cell.column );

      if ( field == LINK_DATE )
        xiev->refused = !validate_date_char( xiev->v.chr.ch );
      break;
    }
    ...
  }
}
```

XIE_CHAR_FIELD Character for Edit Field

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_chr
    {
      struct _xi_obj *xi_obj;
      int ch;
      BOOLEAN shift;
      BOOLEAN control;
      BOOLEAN is_paste;
    } chr;
    ...
  } v;
} XI_EVENT;
```

Description

This event occurs whenever an edit field has focus and a character is typed for that edit field or a paste operation is done on that edit field. This event allows you to modify and/or validate the character before it is inserted into the current text for the edit field. Movement characters, such as left and right arrows, home and end, also generate character events. If the event is refused for these movement characters, the movement will not occur.

This event uses the **chr** member of the union **v** in **XI_EVENT**.

xi_obj is the edit field object that has focus and is receiving a character.

ch is the character. This value may be modified. If so, that new character will be put into the edit field text instead of the original one. For example, you may wish to force lower case to upper case.

shift is **TRUE** if the shift key was being held when the character was typed.

control is **TRUE** if the control key was being held when the character was typed.

is_paste is **TRUE** if the character is the result of a paste command, rather than an actual character typed on the keyboard. If this value is **TRUE**, you may want to avoid beeps, dialogs, etc. because there will usually be many character events in a row during a paste operation.

Refusable

This event can be refused. If it is refused, the character will not be inserted into the edit field text or, if the character would move the insertion point, that movement does not occur.

Example

The following code is from “lstlink.c”.

```
static BOOLEAN validate_date_char( int ch )
{
  if ( ch > 255 || !isprint( ch ) )
    return TRUE;
  if ( ch == '/' || ch == '-' || isdigit( ch ) )
    return TRUE;
  return FALSE;
}

static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_CHAR_FIELD:
      if ( xiev->v.chr.xi_obj->cid == FIELD_BASE_CID + LINK_DATE )
        xiev->refused = !validate_date_char( xiev->v.chr.ch );
      break;
    ...
  }
}
```

XIE_CHG_CELL Cell Changed

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event occurs whenever there is a change to the text for a cell. This event may occur several times while a cell has focus.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj contains the cell object which changed.

Refusable

This event cannot be refused. Changes can be prevented by refusing the **XIE_CHAR_CELL** event.

Example

The following code is from “lstdb.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_CHG_CELL:
    {
      LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

      list_info->cell_changed = TRUE;
      break;
    }
    ...
  }
}
```

XIE_CHG_FIELD Edit Field Changed

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event occurs whenever there is a change to the text for an edit field. This event may occur several times while an edit field has focus.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj contains the edit field object which changed.

Refusable

This event cannot be refused. Changes can be prevented by refusing the **XIE_CHAR_FIELD** event.

Example

The following code is from “lstdb.c”.

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_CHG_FIELD:
      form_info->field_changed = TRUE;
      break;
    ...
  }
}
```

XIE_CLEANUP Final Event for an Interface

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent after an interface has been closed, but before the memory for the objects has been freed. The application should free any data stored in the application data of the interface.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is interface object being closed.

Refusable

This event cannot be refused. The closing of a window can be prevented by refusing the **XIE_CLOSE** event.

Example

The following code is from “lstdb.c”.

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_CLEANUP:
      if ( form_info->allocated )
      {
        XI_EVENT xiev;

        xiev.v.rec_free.record = form_info->handle;
        emp_free( &xiev );
      }
      break;
    ...
  }
}
```

XIE_CLOSE Close Window Request

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the user attempts to close the window containing the interface.

This event is not sent when the application closes the window by calling **xi_delete** on the interface object. Rather, it is sent only when the user clicks on the close box of the window, or selects close from the window system menu. Any action that must be taken when the window is actually closed should be taken upon the **XIE_CLEANUP** event.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is interface object being closed.

Refusable

This event can be refused. If it is refused, the interface will not be closed.

Example

The following code is from "lstdb.c".

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_CLOSE:
      if ( ( !xi_move_focus( itf ) || form_info->changed )
         && xvt_ask( "No", "Yes", NULL,
         "You have made changes. Are you sure you want to cancel?" )
         != RESP_2 )
        xiev->refused = TRUE;
      break;
    ...
  }
}
```

XIE_COL_DELETE Column Delete

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_column
    {
      struct _xi_obj *list;
      int   col_nbr;
      int   new_col_nbr;
      BOOLEAN in_fixed;
      int   new_col_width;
    } column;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the user attempts to delete a column by dragging and dropping the column off of the list.

This event uses the **column** member of the union **v** in **XI_EVENT**.

list is the list that contains the column being deleted.

col_nbr is the column number of the column being deleted. Since columns may be moved, you should retrieve the column ID, by accessing the **XI_OBJ** structure for the column, for example:

```
switch ( xiev->v.column.list->children[
         xiev->v.column.col_nbr ]->cid )
{
  ...
```

new_col_nbr, **in_fixed**, and **new_col_width** are not used by this event.

Refusable

This event can be refused. If it is refused, the column will not be deleted from the list.

Example

The following code is from “lstmem.c”.

```
static BOOLEAN do_col_delete( LIST_INFO* list_info, XI_EVENT* xiev )
{
  int                count;
  struct xit_column* column = &xiev->v.column;
  XI_OBJ**           members = xi_get_member_list( column->list,
                                                   &count );
  XI_OBJ_DEF*        column_def;

  column_def = xi_get_def( members[column->col_nbr] );
  add_column_def( &list_info->deleted_column_list, column_def );
  return TRUE;
}

static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_COL_DELETE:
      xiev->refused = !do_col_delete( list_info, xiev );
      break;
    ...
  }
}
```

XIE_COL_MOVE Column Move

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_column
    {
      struct _xi_obj *list;
      int   col_nbr;
      int   new_col_nbr;
      BOOLEAN in_fixed;
      int   new_col_width;
    } column;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the user attempts to move a column by dragging and dropping the column.

This event uses the **column** member of the union **v** in **XI_EVENT**.

list is the list that contains the column being moved.

col_nbr is the column number of the column being moved. Since columns may move, you should retrieve the column ID, by accessing the **XI_OBJ** structure for the column, for example:

```
switch ( xiev->v.column.list->children[
         xiev->v.column.col_nbr ]->cid )
{
  ...
```

new_col_nbr is the column number where the column will be after the move.

in_fixed will be set to **TRUE** if the column is being moved into the fixed portion of a horizontally scrolling list.

new_col_width is not used by this event.

Refusable

This event can be refused. If it is refused, the column will not be moved.

XIE_COL_SIZE Column Resize

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_column
    {
      struct _xi_obj *list;
      int   col_nbr;
      int   new_col_nbr;
      BOOLEAN in_fixed;
      int   new_col_width;
    } column;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the user attempts to resize a column by dragging the border between the column headings.

This event uses the **column** member of the union **v** in **XI_EVENT**.

list is the list that contains the column being resized.

col_nbr is the column number of the column being resized. Since columns may be moved, you should retrieve the column ID, by accessing the **XI_OBJ** structure for the column, for example:

```
switch ( xiev->v.column.list->children[
         xiev->v.column.col_nbr ]->cid )
{
  ...
```

new_col_width is the new column width, in form units.

new_col_nbr and **in_fixed** are not used by this event.

Refusable

This event can be refused. If it is refused, the column will not be resized.

Example

The following code is from “lstdb.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_COL_SIZE:
    {
      LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

      list_info->list_resizing = TRUE;
      break;
    }
```

```
    case XIE_XVT_POST_EVENT:
      if ( xiev->v.xvte.type == E_MOUSE_UP )
      {
        LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

        if ( list_info->list_resizing )
        {
          RCT    rct;
          WINDOW win = xi_get_window( itf );

          xi_get_rect( itf, &rct );
          xvt_vobj_translate_points( win,
xvt_vobj_get_parent( win ),
                                     (PNT*)&rct, 2 );
          xvt_vobj_move( win, &rct );
        }
      }
      break;
    ...
  }
}
```

XIE_COMMAND Menu Item Selected

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_cmd
    {
      int tag;
      BOOLEAN shift;
      BOOLEAN control;
    } cmd;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when a menu item is selected.

This event uses the **cmd** member of the union **v** in **XI_EVENT**.

tag is the menu tag of the selected menu item.

shift is **TRUE** if the shift key was being held down when the menu item was selected.

control is **TRUE** if the control key was being held down when the menu item was selected.

Refusable

This event cannot be refused. However, if you do not process the event, nothing will happen as a result of the menu selection. The one exception is that XI may process cut, copy and paste commands. You can refuse these commands by refusing the XVT **E_COMMAND** event from the **XIE_XVT_EVENT**.

Example

The following code is from “lstdb.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_COMMAND:
      do_menu( xiev->v.cmd.tag );
      break;
    ...
  }
}
```

The **do_menu** function is from “main.c”.

```
void do_menu( MENU_TAG cmd )
{
  switch ( cmd )
  {
    case M_FILE_QUIT:
      xvt_terminate();
      break;
    case M_VIEW_MEMORY:
      open_memory_list( 64 );
      break;
    case M_VIEW_EMPLOYEE:
      open_employee_list();
      break;
    case M_VIEW_LINKED_LIST:
      open_linked_list();
      break;
    case M_VIEW_SYNC_LIST:
      open_synchronized_list();
      break;
  }
}
```

XIE_DBL_CELL Double Click on a Cell

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the user double clicks on a cell that is not selectable. If the cell is selectable, an **XIE_SELECT** event is generated with **dbl_click** set to **TRUE**.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the cell object that was double clicked.

Note that you should never call a function such as **xvt_note** upon receiving this event. When your application gets this event, the mouse is down and trapped in the window where the cell is. However, there is a work around, as follows:

1. Keep track of the state of the mouse: i.e. is the mouse down or up.
2. Make your own function to put up a note box. In this function, if the mouse is down, simply flag that the note needs to be put up, and return. If the mouse is up, put up the note.
3. On every mouse up event, call a function in the same module as your special note function. If there is a note box waiting to be put up, then put it up.

Refusable

This event cannot be refused. However, if you do nothing in this event, the double click will have no effect other than selecting a word in the text.

Example

The following code is from “lstlink.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_DBL_CELL:
    {
      XI_OBJ* cell = xiev->v.xi_obj;

      open_form( cell->parent, row_to_record( cell->parent,
                 cell->v.cell.row ) );
      break;
    }
    ...
  }
}
```

XIE_DBL_FIELD Double Click on an Edit Field

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the user double clicks on an edit field.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the edit field object that was double clicked.

Note that you should never call a function such as **xvt_note** upon receiving this event. When your application gets this event, the mouse is down and trapped in the window where the edit field is.

See **XIE_DBL_CELL** for information about putting up note boxes upon this event.

Refusable

This event cannot be refused. However, if you do nothing in this event, the double click will have no effect other than selecting a word in the text.

XIE_DROP_ROW Row dropped

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
 union
  {
    ...
    struct xit_drop_row
    {
      struct _xi_obj* src_list;
      struct _xi_obj* dest_list;
      long src_rec;
      long dest_rec;
      BOOLEAN after_all_rows;
      BOOLEAN delete_row;
      BOOLEAN shift;
      BOOLEAN control;
    } drop_row;
    ...
  } v;
} XI_EVENT;
```

Description

This event uses the drop_row member of the union **v** in **XI_EVENT**.

src_list is the list object that the row originated from.

dest_list is the list object that the row is being dropped on.

src_rec contains the handle of the row from the source list.

dest_rec contains the handle of the row in the destination list.

after_all_rows indicates if the row is being dropped at the bottom of the src_list.

delete_row indicates if the row is being deleted.

shift indicates that the shift key was held down when the row was dropped.

control indicates that the control key was held down when the row was dropped.

Refusable

This event can be refused. If it is refused, the row will not move.

Example

The following code is from “lstlink.c”.

```
case XIE_DROP_ROW:
      {
        if ( xiev->v.drop_row.src_list->cid != LIST_CID )
        {
          xvt_dm_post_note( "Invalid drag and drop attempted." );
          break;
        }
        if ( xiev->v.drop_row.delete_row )
        {
          if ( xvt_dm_post_ask( "No", "Yes", NULL,
               "Are you sure you want to delete all selected
rows?" )
               == RESP_2 )
            delete_selected( xiev->v.drop_row.src_list );
        }
        else
        {
          long dest_rec;
          LIST_INFO *src_list_info;
          LIST_INFO *list_info =
             ( LIST_INFO * ) xi_get_app_data( itf );

          if ( xiev->v.drop_row.after_all_rows )
            dest_rec = 0;
          else
            dest_rec = xiev->v.drop_row.dest_rec;
          if ( xiev->v.drop_row.src_list ==
                  xiev->v.drop_row.dest_list
               && xiev->v.drop_row.src_rec == dest_rec )
            break;
          src_list_info = ( LIST_INFO * ) xi_get_app_data(
                                  xiev->v.drop_row.src_list-
>parent );
          link_copy_selected( src_list_info->link_list,
                              list_info->link_list,
                              xiev->v.drop_row.dest_rec );
          if ( !xiev->v.drop_row.control )
            delete_selected( xiev->v.drop_row.src_list );
          if ( xiev->v.drop_row.control
               || xiev->v.drop_row.src_list
               != xiev->v.drop_row.dest_list )
            refresh_list( xiev->v.drop_row.dest_list );
        }
        break;
```

XIE_GET_FIRST First Record Request for a List

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_rec_request
    {
      struct _xi_obj *list;
      long spec_rec;
      long data_rec;
      short percent;
      unsigned long attrib;
      XinColor color;
      int row_height;
      BOOLEAN has_focus;
    } rec_request;
    ...
  } v;
} XI_EVENT;
```

Description

This event uses the **rec_request** member of the union **v** in **XI_EVENT**.

list is the list object that the record is being requested for.

data_rec should be set to the appropriate record handle, unless you are responding to **XIE_REC_ALLOCATE** events, in which case, you will use the value already set in **data_rec**. The appropriate record handle depends upon the value in **percent**. So, if **percent** is zero, the record handle will refer to the first record. If **percent** is fifty, the record handle should refer to a record halfway through the total number of records. If there is no vertical scroll bar on the list, **percent** will always be zero.

attrib is a bitwise OR'ed combination of **XI_ATR_*** values which control some of the behavior of the row.The following attributes apply to rows:

```
XI_ATR_SELECTED
```

color set the default foreground color for text in this row.

row_height sets the height of the row. If the column has **wrap_text** enabled, you should not change this value. By default, rows are just tall enough for the cell text.

has_focus is only used if **XI_PREF_KEEP_FOCUS_FIXED** is set to **TRUE**. In that case, XI must know if the row with focus has been scrolled back into the visible region of the list after being scrolled off. If the record handle stays the same, this is sufficient information. However, if you are allocating and freeing record handles, you will need to set this value to **TRUE** so that XI knows to show focus on this row as it scrolls back into the visible region.

spec_rec is not used by this event.

Refusable

This event can be refused. If it is refused, the list will be empty.

Example

The following code is from "lstlink.c".

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_GET_FIRST:
    {
      LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

      xiev->refused = !link_get_first( list_info->link_list, xiev );
      break;
    }
    ...
  }
}
```

The **link_get_first** function is from "datlink.c".

```
BOOLEAN link_get_first( LINKED_LIST* link_list, XI_EVENT* xiev )
{
  long rec_num;

  if ( link_list->nbr_records == 0 )
    return FALSE;

  rec_num = link_list->nbr_records * xiev->v.rec_request.percent
            / 100;
  xiev->v.rec_request.data_rec = PTR_LONG( find_record( link_list,
                                 rec_num ) );
  xiev->v.rec_request.attrib = get_attribute(
                               xiev->v.rec_request.data_rec );
  return TRUE;
}
```

XIE_GET_LAST Last Record Request for a List

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_rec_request
    {
      struct _xi_obj *list;
      long spec_rec;
      long data_rec;
      short percent;
      unsigned long attrib;
      XinColor color;
      int row_height;
      BOOLEAN has_focus;
    } rec_request;
    ...
  } v;
} XI_EVENT;
```

Description

This event uses the **rec_request** member of the union **v** in **XI_EVENT**.

list is the list object that the record is being requested for.

data_rec should be set to the record handle that corresponds to the last record, unless you are responding to **XIE_REC_ALLOCATE** events, in which case, you will use the value already set in **data_rec**.

attrib is an OR'ed combination of **XI_ATR_*** values which control some of the behavior of the row.The following attributes apply to rows:

```
XI_ATR_SELECTED
```

color set the default foreground color for text in this row.

row_height sets the height of the row. If the column has **wrap_text** enabled, you should not change this value. By default, rows are just tall enough for the cell text.

has_focus is only used if **XI_PREF_KEEP_FOCUS_FIXED** is set to **TRUE**. In that case, XI must know if the row with focus has been scrolled back into the visible region of the list after being scrolled off. If the record handle stays the same, this is sufficient information. However, if you are allocating and freeing record handles, you will need to set this value to **TRUE** so that XI knows to show focus on this row as it scrolls back into the visible region.

spec_rec and **percent** are not used by this event.

Refusable

This event can be refused. If it is refused, the list will be empty.

See Also

XIE_GET_FIRST, **XIE_GET_NEXT**, **XIE_GET_PREV**

Example

The following code is from “lstlink.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_GET_LAST:
    {
      LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

      xiev->refused = !link_get_last( list_info->link_list, xiev );
      break;
    }
    ...
  }
}
```

The **link_get_last** function is from “datlink.c”.

```
BOOLEAN link_get_last( LINKED_LIST* link_list, XI_EVENT* xiev )
{
  if ( link_list->nbr_records == 0 )
    return FALSE;
  xiev->v.rec_request.data_rec = PTR_LONG( ((REC_INFO*)link_list
                                  ->first_rec)->prev );
  xiev->v.rec_request.attrib = get_attribute(
                                xiev->v.rec_request.data_rec );
  return TRUE;
}
```

XIE_GET_NEXT — Next Record Request for a List

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_rec_request
    {
      struct _xi_obj *list;
      long spec_rec;
      long data_rec;
      short percent;
      unsigned long attrib;
      XinColor color;
      int row_height;
      BOOLEAN has_focus;
    } rec_request;
    ...
  } v;
} XI_EVENT;
```

Description

This event uses the **rec_request** member of the union **v** in **XI_EVENT**.

list is the list object that the record is being requested for.

spec_rec contains a record handle that was returned by a previous **XIE_GET_*** event.

data_rec should be set to the record handle corresponding to the record immediately following the record specified by **spec_rec**, unless you are responding to **XIE_REC_ALLOCATE** events, in which case, you will use the value already set in **data_rec**.

attrib is an OR'ed combination of **XI_ATR_*** values which control some of the behavior of the row.The following attributes apply to rows:

```
XI_ATR_SELECTED
```

color set the default foreground color for text in this row.

row_height sets the height of the row. If the column has **wrap_text** enabled, you should not change this value. By default, rows are just tall enough for the cell text.

has_focus is only used if **XI_PREF_KEEP_FOCUS_FIXED** is set to **TRUE**. In that case, XI must know if the row with focus has been scrolled back into the visible region of the list after being scrolled off. If the record handle stays the same, this is sufficient information. However, if you are allocating and freeing record handles, you will need to set this value to **TRUE** so that XI knows to show focus on this row as it scrolls back into the visible region.

percent is not used by this event.

Refusable

This event can be refused. It should be refused if there is no record following **spec_rec**.

See Also

XIE_GET_FIRST, **XIE_GET_LAST**, **XIE_GET_PREV**

Example

The following code is from “lstlink.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_GET_NEXT:
      xiev->refused = !link_get_next( xiev );
      break;
    ...
  }
}
```

The **link_get_next** function is from “datlink.c”.

```
BOOLEAN link_get_next( XI_EVENT* xiev )
{
  REC_INFO* rec = (REC_INFO*)xiev->v.rec_request.spec_rec;

  if ( rec->rec_nbr == rec->link_list->nbr_records - 1 )
    return FALSE;
  xiev->v.rec_request.data_rec = PTR_LONG( rec->next );
  xiev->v.rec_request.attrib = get_attribute(
                               xiev->v.rec_request.data_rec );
  return TRUE;
}
```

XIE_GET_PERCENT Request for Percentage Through List Records

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_get_percent
    {
      struct _xi_obj *list;
      long record;
      short percent;
    } get_percent;
    ...
  } v;
} XI_EVENT;
```

Description

This event requests the percentage through the total list of records of the specified record handle. The percentage returned is used to set the position of the scroll bar thumb. On some platforms, the size (proportion) of the thumb is also set based on the results of this event. These events only occur if the list has a vertical scroll bar.

Every time the vertical scroll bar must be updated, XI sends two **XIE_GET_PERCENT** events. The first event occurs for the first fully visible record in the list and the second event occurs for the last fully visible row in the list. The actual thumb position and proportion is calculated from those two percentages.

This event uses the **get_percent** member of the union **v** in **XI_EVENT**.

list is the list object that the percentage is being requested for.

record is a record handle that came from an earlier **XIE_GET_*** event.

percent should be set to the correct percentage for that record handle.

Refusable

This event cannot be refused.

Example

The following code is from “lstlink.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_GET_PERCENT:
      link_get_percent( xiev );
      break;
    ...
  }
}
```

The **link_get_percent** function is from “datlink.c”.

```
void link_get_percent( XI_EVENT* xiev )
{
  REC_INFO* rec = (REC_INFO*)xiev->v.get_percent.record;

  if ( rec->link_list->nbr_records > 1 )
  {
    xiev->v.get_percent.percent = (int)( rec->rec_nbr * 100
                                  / ( rec->link_list->nbr_records
                                  - 1 ) );
    if ( xiev->v.get_percent.percent == 0 && rec->rec_nbr != 0 )
      xiev->v.get_percent.percent = 1;
  }
}
```

XIE_GET_PREV Previous Record Request for a List

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_rec_request
    {
      struct _xi_obj *list;
      long spec_rec;
      long data_rec;
      short percent;
      unsigned long attrib;
```

```
      XinColor color;
      int row_height;
      BOOLEAN has_focus;
    } rec_request;
    ...
  } v;
} XI_EVENT;
```

Description

This event uses the **rec_request** member of the union **v** in **XI_EVENT**.

list is the list object that the record is being requested for.

spec_rec contains a record handle that was returned by a previous **XIE_GET_*** event.

data_rec should be set to the record handle corresponding to the record immediately before the record specified by **spec_rec**, unless you are responding to **XIE_REC_ALLOCATE** events, in which case, you will use the value already set in **data_rec**.

attrib is an OR'ed combination of **XI_ATR_*** values which control some of the behavior of the row.The following attributes apply to rows:

```
XI_ATR_SELECTED
```

color set the default foreground color for text in this row.

row_height sets the height of the row. If the column has **wrap_text** enabled, you should not change this value. By default, rows are just tall enough for the cell text.

has_focus is only used if **XI_PREF_KEEP_FOCUS_FIXED** is set to **TRUE**. In that case, XI must know if the row with focus has been scrolled back into the visible region of the list after being scrolled off. If the record handle stays the same, this is sufficient information. However, if you are allocating and freeing record handles, you will need to set this value to **TRUE** so that XI knows to show focus on this row as it scrolls back into the visible region.

percent is not used by this event.

Refusable

This event can be refused. It should be refused if there is no record previous to **spec_rec**.

See Also

XIE_GET_FIRST, **XIE_GET_LAST**, **XIE_GET_NEXT**

Example

The following code is from "lstlink.c".

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
```

```
  {
    ...
    case XIE_GET_PREV:
      xiev->refused = !link_get_prev( xiev );
      break;
    ...
  }
}
```

The **link_get_prev** function is from “datlink.c”.

```
BOOLEAN link_get_prev( XI_EVENT* xiev )
{
  REC_INFO* rec = (REC_INFO*)xiev->v.rec_request.spec_rec;

  if ( rec->rec_nbr == 0 )
    return FALSE;
  xiev->v.rec_request.data_rec = PTR_LONG( rec->prev );
  xiev->v.rec_request.attrib = get_attribute(
                               xiev->v.rec_request.data_rec );
  return TRUE;
}
```

XIE_INIT Initial Event for an Interface

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This is the first event that occurs for an interface. When this event occurs, the controls have all been created. This allows you to further initialize those controls. For example, you could set the initial text for all the edit fields in the interface.

On some platforms, when using a mult-line edit field, XVT makes it impossible to guarantee that this is the first event. We will fix this in a future release.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the interface being initialized.

Refusable

This event cannot be refused.

Example

The following code is from “lstdb.c”.

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    case XIE_INIT:
    {
      int      num;
      XI_OBJ*  form;
      XI_OBJ** field;

      form = xi_get_obj( itf, FORM_CID );
      field = form->children;
      for ( num = 0; num < form->nbr_children; num++, field++ )
        set_field_text( *field, form_info->handle );
      break;
    }
    ...
  }
}
```

XIE_OFF_CELL Off Cell Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the cell is about to lose the focus. Typically, this event is used to validate the text entered into the cell.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the cell object that is losing focus**.**

Refusable

This event can be refused. If it is refused, focus will not move off of the cell.

See Also

XIE_OFF_FIELD for notes on handling **XIE_OFF_*** events.

Example

The following code is from “lstlink.c”.

```
static BOOLEAN store_cell( LIST_INFO* list_info, XI_OBJ* cell )
{
  LINK_FIELD field = column_to_field( cell->parent,
                                      cell->v.cell.column );
  char*      text = xi_get_text( cell, NULL, 0 );

  if ( !link_validate( field, text ) )
    return FALSE;
  link_set_value( row_to_record( cell->parent, cell->v.cell.row ),
                  field, text );
  list_info->cell_changed = FALSE;
  xi_cell_request( cell );
  return TRUE;
}

static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_OFF_CELL:
    {
      LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

      if ( list_info->cell_changed )
        xiev->refused = !store_cell( list_info, xiev->v.xi_obj );
      break;
    }
    ...
  }
}
```

XIE_OFF_COLUMN Off Column Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving from a cell in one column to a cell in another column or to some other control in the same interface that is not part of the list.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the column object that is losing focus**.**

Refusable

This event can be refused. If it is refused, focus will not move off of the cell in this column.

See Also

XIE_OFF_FIELD for notes on handling **XIE_OFF_*** events.

XIE_OFF_FIELD Off Edit Field Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the edit field is about to lose the focus. Typically, this event is used to validate the text entered into the edit field.

You should not put up a note box or modal dialog box upon the receipt of any **XIE_OFF_*** event because the focus change may be caused by a mouse click, and the mouse would probably still be down. This would cause abnormal behavior because the mouse up event would go to the note box or dialog box, and XI would never receive it, and think that the mouse is still down.

Rather, if you want to give advice, have a readonly edit field or a static text control reserved for this purpose, and set the text of the edit field or static text control to an error message when the user enters invalid data.

Another approach is to do a work around for the problem with the trapped mouse, as follows:

1. Keep track of the state of the mouse: i.e. is the mouse down or up.
2. Make your own function to put up a note box. In this function, if the mouse is down, simply flag that the note needs to be put up, and return. If the mouse is up, put up the note.
3. On every mouse up event, call a function in the same module as your special note function. If there is a note box waiting to be put up, then put it up.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the edit field object that is losing focus**.**

Refusable

This event can be refused. If it is refused, focus will not move off of the edit field.

See Also

XIE_OFF_FIELD for notes on handling **XIE_OFF_*** events.

Example

The following code is from “lstdb.c”.

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_OFF_FIELD:
      if ( form_info->field_changed )
      {
        XI_OBJ* field = xiev->v.xi_obj;

        if ( !emp_set_value( form_info->handle,
                             field->cid - FIELD_BASE_CID,
                             xi_get_text( field, NULL, 0 ) ) )
        {
          xiev->refused = TRUE;
          xi_set_sel( field, 0, INT_MAX );
        } else
        {
          form_info->changed = TRUE;
          form_info->field_changed = FALSE;
          set_field_text( field, form_info->handle );
        }
      }
      break;
  }
}
```

XIE_OFF_FORM Off Form Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving from an edit field in this form to a control that is not in this form, but is in the same interface.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the form object that is losing focus**.**

Refusable

This event can be refused. If it is refused, focus will not move off of the edit field in this form.

See Also

XIE_OFF_FIELD for notes on handling **XIE_OFF_*** events.

XIE_OFF_GROUP Off Group Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving from a control in a group to a control that is not in that group.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the group object that is losing focus**.**

Refusable

This event can be refused. If it is refused, focus will not move off of the control in this group.

See Also

XIE_OFF_FIELD for notes on handling **XIE_OFF_*** events.

Example

The following code is from “lstdb.c”.

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_OFF_GROUP:
      if ( form_info->changed && !emp_validate_hours(
           form_info->handle ) )
      {
        xvt_error( "Minimum hours cannot be greater than
maximum." );
        xiev->refused = TRUE;
      }
      break;
    ...
  }
}
```

XIE_OFF_LIST Off List Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving from a cell in one list to a cell in another list or to some other control that is not part of the list, but is in the same interface.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the list object that is losing focus**.**

Refusable

This event can be refused. If it is refused, focus will not move off of the cell in this list.

See Also

XIE_OFF_FIELD for notes on handling **XIE_OFF_*** events.

XIE_OFF_ROW Off Row Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving from a cell in one row to a cell in another row or to some other control that is not part of the list, but is in the same interface. Typically, this event is used to validate entire rows and/or to update the underlying data structure or database to match the current row values.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the row object that is losing focus**.**

Refusable

This event can be refused. If it is refused, focus will not move off of the cell in this row.

See Also

XIE_OFF_FIELD for notes on handling **XIE_OFF_*** events.

Example

The following code is from “lstdb.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_OFF_ROW:
    {
      LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

      if ( list_info->row_changed )
      {
        XI_OBJ* row = xiev->v.xi_obj;

        emp_update( row_to_record( row->parent,
                    row->v.row_data.row ) );
        list_info->row_changed = FALSE;
      }
      break;
    }
    ...
  }
}
```

XIE_ON_CELL On Cell Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the cell is about to gain the focus.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the cell object that is gaining focus**.**

Refusable

This event can be refused. If it is refused, focus will not move onto the cell, but will instead remain where it was.

XIE_ON_COLUMN On Column Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving to a cell in one column from a cell in another column or from some other control in the same interface that is not part of the list.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the column object that is gaining focus**.**

Refusable

This event can be refused. If it is refused, focus will not move onto the cell in this column, but will instead remain where it was.

XIE_ON_FIELD On Edit Field Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the edit field is about to gain the focus.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the edit field object that is gaining focus**.**

Refusable

This event can be refused. If it is refused, focus will not move onto the edit field, but will instead remain where it was.

XIE_ON_FORM On Form Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving to an edit field in this form from a control that is not in this form, but is in the same interface.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the form object that is gaining focus**.**

Refusable

This event can be refused. If it is refused, focus will not move onto an edit field within the form, but will instead remain where it was.

XIE_ON_GROUP On Group Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving to a control in a group from a control that is not in that group.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the group object that is gaining focus**.**

Refusable

This event can be refused. If it is refused, focus will not move onto the control in this group, but will instead remain where it was.

XIE_ON_LIST On List Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving to a cell in one list from a cell in another list or from some other control that is not part of the list, but is in the same interface.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the list object that is gaining focus**.**

Refusable

This event can be refused. If it is refused, focus will not move onto the cell in this list, but will instead remain where it was.

XIE_ON_ROW On Row Focus Change

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct _xi_obj *xi_obj;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when focus is moving to a cell in one row from a cell in another row or from some other control that is not part of the list, but is in the same interface.

This event uses the **xi_obj** member of the union **v** in **XI_EVENT**.

xi_obj is the row object that is gaining focus**.**

Refusable

This event can be refused. If it is refused, focus will not move onto the cell in this row, but will instead remain where it was.

XIE_REC_ALLOCATE Allocate a Record

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_rec_allocate
    {
      struct _xi_obj *list;
      long record;
    } rec_allocate;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent whenever XI is about to send a record request event (**XI_GET_***). It allows your application to allocate a data structure that will be used as the **data_rec** handle for the record request event. If you allocate structures, you will probably want to respond to the **XIE_REC_FREE** event also.

This event uses the **rec_allocate** member of the union **v** in **XI_EVENT**.

list is the list object that the record will be used for.

record is the record handle that you should set to the allocated data structure.

Refusable

This event is cannot be refused. However, if you do not respond to this event, the only result will be that **data_rec** will be zero when the record request event occurs.

Example

The following code is from “lstdb.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_REC_ALLOCATE:
      emp_allocate( xiev );
      break;
    ...
  }
}
```

The **emp_allocate** is from “datdb.c”.

```
void emp_allocate( XI_EVENT* xiev )
{
  xiev->v.rec_allocate.record = PTR_LONG( xvt_mem_alloc(
                                sizeof( EMPREC ) ) );
}
```

XIE_REC_FREE Free a Record

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_rec_free
    {
      struct _xi_obj *list;
      long record;
    } rec_free;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent whenever XI is done with a particular record handle. It allows your application to free any memory that was allocated in the **XIE_REC_ALLOCATE** event.

This event uses the **rec_free** member of the union **v** in **XI_EVENT**.

list is the list object that the record was allocated for.

record is the record handle that needs to be freed, if it was allocated in the **XIE_REC_ALLOCATE** event.

Refusable

This event cannot be refused. However, you can ignore this event if you are not responding to **XIE_REC_ALLOCATE** events.

Example

The following code is from “lstdb.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_REC_FREE:
      emp_free( xiev );
      break;
    ...
  }
}
```

The **emp_free** is from “datdb.c”.

```
void emp_free( XI_EVENT* xiev )
{
  xvt_mem_free( (DATA_PTR)xiev->v.rec_free.record );
}
```

XIE_ROW_SIZE Row Resize

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_row_size
    {
      struct _xi_obj *xi_obj;
      int new_row_height;
    } row_size;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when the user attempts to resize a row by dragging the border between two rows.

This event uses the **row_size** member of the union **v** in **XI_EVENT**.

xi_obj is the row object being sized. If you need the record handle for the row, you will need to access the **row** member of the object data and then use that as an index into the array returned from the **xi_get_list_info** function.

new_row_height is the new height of the row, in pixels.

Refusable

This event can be refused. If it is refused, the row will not be resized.

Example

The following code is from “lstmem.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_ROW_SIZE:
      mem_set_row_height( row_to_record(
                          xiev->v.row_size.xi_obj->parent,
                          xiev->v.select.xi_obj->v.row_data.row ),
                          xiev->v.row_size.new_row_height );
      break;
    ...
  }
}
```

The **mem_set_row_height** funcion is from “datmem.c”.

```
void mem_set_row_height( REC_INFO* rec, int height )
{
  rec->row_height = height;
}
```

XIE_SELECT Select Event

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_select
    {
      struct _xi_obj *xi_obj;
      BOOLEAN selected;
      BOOLEAN dbl_click;
      BOOLEAN shift;
      BOOLEAN control;
      int column;
      long *records;
    } select;
    ...
  } v;
} XI_EVENT;
```

Description

This event is sent when a user selects or deselects a row, column, or range of cells in a list. Row selection is enabled for cells in a column by setting **XI_ATR_SELECTABLE** for that column. Column selection is enabled by setting **XI_ATR_COL_SELECTABLE** for that column. Cell selection is enabled by setting the **select_cells** member of **XI_LIST_DEF** to **TRUE** before the list is created.

This event uses the **select** member of the union **v** in **XI_EVENT**.

xi_obj is the row or column object that was selected or deselected. When a range of cells are selected or deselected, a single **XIE_SELECT** event will occur for the list object. You can then use the **xi_get_cell_selection** function to get the cells actually selected.

selected indicates whether the row or column was selected or deselected. **XIE_SELECT** events for a range of cells always set this value to **TRUE** because no “deselect” event occurs.

dbl_click indicates whether the user selected/deselected the row or column via a double click. You would use this value if you wanted your application to take some action when the user double clicked on either the row or column. Double click selections never occur for a range of cells.

shift is **TRUE** if the shift key was being held when the selection occurred.

control is **TRUE** if the control key was being held when the selection occurred.

column is only set when the selection/deselection is for a row. It indicates which column of that row was actually clicked to make the selection. Since columns may be moved, you will probably want to translate this value into the ID number for the column, for example:

```
if ( xiev->v.column.list->children[
     xiev->v.select.column ]->cid == COL1_ID )
{
  ...
```

The array of record handles is returned in **records**. This is the same array that is returned by the function xi_get_list_info.

Refusable

This event can be refused for rows and columns. If it is refused, the row or column will not reverse its selection state. In other words, if a row is selected and you refuse the event, it will remain selected. If a column is not selected and you refuse the event it will still not be selected.

This event cannot be refused for a list (cell selection).

Example

The following code is from "lstmem.c".

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_SELECT:
      switch ( xiev->v.select.xi_obj->type )
      {
        case XIT_ROW:
        {
          XI_OBJ* list = xiev->v.select.xi_obj->parent;
          int row = xiev->v.select.xi_obj->v.row_data.row;

          if ( column_to_code( list, xiev->v.select.column )
               == VALUE_IN_STOCK )
          {
            XI_OBJ cell;

            mem_change_stock( row_to_record( list, row ) );
            XI_MAKE_CELL( &cell, list, row, xiev->v.select.column );
            xiev->refused = TRUE;
            xi_cell_request( &cell );
          } else
            mem_select_row( row_to_record( list, row ),
                            xiev->v.select.selected );
          break;
        }
```

```
        case XIT_LIST:
        {
          XI_CELL_SPEC* cells;
          int count;
          int num;
          XI_OBJ* list = xiev->v.select.xi_obj;

          cells = xi_get_cell_selection( list, &count );
          select_clear( list->itf, FALSE, FALSE );
          for ( num = 0; num < count; num++, cells++ )
            mem_select_cell( row_to_record( list, cells->row ),
                             column_to_code( list, cells->column ),
                             xiev->v.select.selected );
          break;
        }
      }
      break;
    ...
  }
}
```

XIE_UPDATE Update Drawing

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    EVENT xvte;
    ...
  } v;
} XI_EVENT;
```

Description

This event occurs during the XVT **E_UPDATE** event for the interface. It is sent after XI is done doing all of its drawing in the window, including the background. You should respond to this event if you have drawable buttons or if you want to do special drawing on cells, columns or rows. Other types of drawing are also possible (e.g. you may want to draw an icon in an otherwise empty part of the interface).

This event uses the **xvte** member of the union **v** in **XI_EVENT**. This member contains the XVT event information for the current **E_UPDATE** event.

Refusable

This event cannot be refused. If it is ignored, the XI interface will draw normally.

XIE_VIR_PAN Virtual Pan

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    struct xit_vir_pan
    {
      BOOLEAN before_pan;
      int delta_x;
      int delta_y;
    } vir_pan;
    ...
  } v;
} XI_EVENT;
```

Description

This event only occurs for virtual interfaces, that is, interfaces that had **virtual_itf** set to **TRUE** in the interface definition. There will be two of these events whenever the interface is "panned" (scrolled). One occurs before the pan and one after. Typically, the application would use this event to move other controls or objects in the interface that are not managed by XI. For instance, if the application is drawing using XVT drawing functions in a window that contains a virtual interface, then when the interface is panned, the application could take note of the new pixel position of the drawing.

This event uses the **vir_pan** member of the union **v** in **XI_EVENT**.

before_pan is **TRUE** if this is the event before the pan.

delta_x and **delta_y** indicate the distance, in pixels, that the virtual interface is going to be panned or was just panned. If the interface was panned to the right (and the controls move to the left visually), then **delta_x** is a positive value. If the interface was panned to the bottom (and the controls move to the top visually), then **delta_y** is a positive value.

Refusable

This event cannot be refused. However, you can call **xi_vir_pan** to pan to a different part of the interface.

XIE_XVT_EVENT Preprocess XVT Event

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    EVENT xvte;
    ...
  } v;
} XI_EVENT;
```

Description

This event occurs for every XVT event that is sent to the interface. This event is sent before XI processes the XVT event. The **XIE_XVT_POST_EVENT** will occur after XI processes the XVT event.

When your application needs to do XVT programming in a window that also contains an XI interface, it would often need to respond to XVT events. By using this event, you can put code that processes XVT events in the same function or module as code that processes XI events.

This event uses the **xvte** member of the union **v** in **XI_EVENT**. This member contains the information from the XVT event.

Refusable

This event can be refused. If it is refused, XI will not process the XVT event. For example, if an **E_MOUSE_DOWN** event would cause focus to move to an edit field, refusing this event would prevent XI from trying to move focus, generating on and off focus events, etc.

Example

The following code is from “lstmem.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    case XIE_XVT_EVENT:
      if ( xiev->v.xvte.type == E_FONT )
      {
        if ( !xvt_font_is_mapped( xiev->v.xvte.v.font.font_id ) )
          xvt_font_map( xiev->v.xvte.v.font.font_id,
                        xi_get_window(itf) );
        select_set_font( itf, xiev->v.xvte.v.font.font_id );
      }
      break;
    ...
  }
}
```

XIE_XVT_POST_EVENT — Post Process XVT Event

Summary

```
typedef struct _xi_event
{
  XI_EVENT_TYPE type;
  BOOLEAN refused;
  union
  {
    ...
    EVENT xvte;
    ...
  } v;
} XI_EVENT;
```

Description

This event occurs for every XVT event that is sent to the interface. This event is sent after XI processes the XVT event. The **XIE_XVT_ EVENT** occurs before XI processes the XVT event.

One use for this function is to do processing after XI has finished completely processing an XVT event. For example, a focus change may cause multiple on and off focus events and may event cause record and cell requests if a list is scrolling. You may want to set a flag, and then read that flag during this event to perform some action after all of the XI events are complete.

This event uses the **xvte** member of the union **v** in **XI_EVENT**. This member contains the information from the XVT event.

Refusable

This event cannot be refused. You can refuse XVT events before XI processes them by refusing the **XIE_XVT_EVENT**.

Example

The following code is from “lstdb.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_XVT_POST_EVENT:
      if ( xiev->v.xvte.type == E_MOUSE_UP )
      {
        LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

        if ( list_info->list_resizing )
        {
          RCT    rct;
          WINDOW win = xi_get_window( itf );

          xi_get_rect( itf, &rct );
          xvt_vobj_translate_points( win,
xvt_vobj_get_parent( win ),
                                     (PNT*)&rct, 2 );
          xvt_vobj_move( win, &rct );
        }
      }
      break;
  }
}
```

Chapter 3

XI Functions

xi_add_button_def — Add Button Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_button_def( XI_OBJ_DEF *parent, int cid,
                               XinRect *rct, unsigned long attrib,
                               char *text, int tab_cid );
```

Description

This function allocates and initializes a button definition. In addition, it adds it to the list of children for the specified parent object definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free or xi_def_free**.

parent may point to either an interface definition or a button container definition. It can also be **NULL** which implies that you will specify an existing interface or button container object in the call to **xi_create**. In that case, the button will be created and added to that existing object.

cid contains the control ID for the button.

rct points to the bounding rectangle of the button. If **parent** points to a container, **rct** will be ignored and can be **NULL**.

attrib is a bitwise OR'ed combination of **XI_ATR_*** values. The following attributes apply to buttons:

```
XI_ATR_ENABLED
XI_ATR_VISIBLE
```

text contains the initial title of the button. The title of the button object may be changed after creation with the **xi_set_text** function.

tab_cid contains the control ID of the control where focus will be moved when the user presses the tab key. On certain platforms, such as the Macintosh, buttons cannot receive the focus, so buttons are left out of the tabbing sequence.

Return Value

Returns a pointer to the allocated button definition.

See Also

XI_OBJ_DEF, **XI_BTN_DEF**

Example

The following code is from "lstdb.c".

```
btndef = xi_add_button_def( cntrdef, ADD_BTN_CID, NULL,
                            XI_ATR_ENABLED | XI_ATR_VISIBLE, "Add",
                            CHG_BTN_CID );
btndef->v.btn->down_icon_rid = ADD_BTN_ICON;
btndef->v.btn->up_icon_rid = ADD_BTN_ICON;
```

xi_add_column_def — Add Column Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_column_def( XI_OBJ_DEF *list, int cid,
                               unsigned long attrib,
                               int sort_number, int width,
                               int text_size,
                               char *heading_text );
```

Description

This function allocates and initializes a column definition. In addition, it adds it to the list of children for the specified list definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

list points to a list definition. It can also be **NULL** which implies that you will specify an existing list object in the call to **xi_create** which will add this column to that list.

cid contains the control ID for the column.

attrib is a bitwise OR'ed combination of **XI_ATR_*** values. The following attributes apply to columns:

```
XI_ATR_ENABLED
XI_ATR_EDITMENU
XI_ATR_AUTOSELECT
XI_ATR_AUTOSCROLL
XI_ATR_FOCUSBORDER
XI_ATR_RJUST
XI_ATR_READONLY
XI_ATR_PASSWORD
XI_ATR_SELECTED
XI_ATR_SELECTABLE
XI_ATR_COL_SELECTABLE
```

sort_number contains an integer number. The columns are initially sorted in ascending **sort_number** order. If more than one column has the same **sort_number**, results are unpredictable.

width is the width of the column, in form units.

text_size is the buffer size of the column which is the number of characters that can be typed into cells in this column. You can change this after the column is created by calling **xi_set_bufsize**. If **var_len_text** is set for the column the buffer size is variable in length.

heading_text contains the initial string for the column heading. You can change the contents of the column heading after the column has been created by calling **xi_set_text** on the column object. The text can contain '\n' characters, making the heading a multi-line heading.

Return Value

Returns a pointer to the allocated column definition.

See Also

XI_OBJ_DEF, **XI_COLUMN_DEF**

Example

The following code is from “lstmem.c”.

```
  coldef = xi_add_column_def( listdef, COL_BASE_CID +
VALUE_ITEM_NBR,
                              STD_COL_ATR | XI_ATR_SELECTABLE,
                              1, 6 * XI_FU_MULTIPLE, 5, "Nbr" );
#if THREE_DIMENSIONAL
  coldef->v.column->heading_platform = TRUE;
  coldef->v.column->column_platform = TRUE;
#endif
  coldef->v.column->size_rows = TRUE;
```

xi_add_container_def Add Container Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_container_def ( XI_OBJ_DEF *itf, int cid,
            XinRect* xi_rct, XI_CONTAINER_ORIENTATION orientation,
            int tab_cid );
```

Description

This function allocates and initializes a button container definition. In addition, it adds it to the list of children for the specified interface definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

A button container is necessary to get proper behavior of radio buttons. The container defines which radio buttons are grouped such that only one of the group can be checked at any time.

A button container is used to provide the bounding rectangle for several buttons. When button containers are used for placing buttons, the buttons will be placed by XI in such a fashion that they will have the correct height and width on every implementation of XVT. Also, a button container specifies how the buttons will be oriented.

itf points to the interface definition. Most likely, the interface definition was previously created via a call to **xi_create_itf_def**. It can also be **NULL** which implies that you will specify an existing interface object in the call to **xi_create** which will add this button container and its children to that interface.

cid contains the control ID for the container to be created.

xi_rct points to a rectangle, in form units, that contains the bounding rectangle for the button container.

orientation specifies whether the contained buttons will be arranged horizontally or vertically. It may be either **XI_STACK_HORIZONTAL**, **XI_STACK_VERTICAL**, **XI_GRID_HORIZONTAL**, or **XI_GRID_VERTICAL**.

tab_cid is the control ID of the form, list, or container that is next in the meta-tabbing sequence. The focus will go to that object when the character defined in the preference **XI_PREF_ITF_TAB_CHAR** is pressed.

The bounding rectangle for any buttons contained in a button container will be ignored.

Return Value

Returns a pointer to the allocated button container definition.

See Also

XI_OBJ_DEF, **XI_CONTAINER_DEF**

Example

The following code is from “lstdb.c”.

```
  rct.top = 0;
  rct.left = 0;
  rct.bottom = 24;
#if XVTWS == WMWS
  rct.right = 192;
  cntrdef = xi_add_container_def( itfdef, CONTAINER_CID, &rct,
                                  XI_STACK_HORIZONTAL, LIST_CID );
#else
  rct.right = 144;
  cntrdef = xi_add_container_def( itfdef, CONTAINER_CID, &rct,
                                  XI_GRID_HORIZONTAL, LIST_CID );
  cntrdef->v.container->packed = TRUE;
#endif
  cntrdef->v.container->btn_width = 6 * XI_FU_MULTIPLE;
```

xi_add_field_def Add Edit Field Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_field_def ( XI_OBJ_DEF *form, int cid,
                               int v, int h, int field_width,
                               unsigned long attrib, int tab_cid,
                               int text_size, XinColor
enabled_color,
                               XinColor back_color,
                               XinColor disabled_color,
                               XinColor disabled_back_color,
                               XinColor active_color );
```

Description

This function allocates and initializes an edit field definition. In addition, it adds it to the list of children for the specified form definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

form points to a form definition. Most likely, the form definition was previously created via a call to **xi_add_form_def**. It can also be **NULL** which implies that you will specify an existing form object in the call to **xi_create** which will add this edit field to that form.

cid contains the control ID for the edit field.

v and **h** are the vertical and horizontal positions, in form units, of the edit field.

field_width is the width of the edit field, in form units.

attrib is a bitwise OR'ed combination of **XI_ATR_*** values. The following attributes apply to edit fields:

```
XI_ATR_ENABLED
XI_ATR_EDITMENU
XI_ATR_AUTOSELECT
XI_ATR_AUTOSCROLL
XI_ATR_FOCUSBORDER
XI_ATR_RJUST
XI_ATR_READONLY
XI_ATR_PASSWORD
XI_ATR_BORDER
XI_ATR_VISIBLE
```

tab_cid contains the ID of the control where focus will be moved when the user presses the tab key.

text_size is the buffer size of the edit field, in characters. You can change this later on a created edit field object by calling **xi_set_bufsize**. If **var_len_text** is set for the field, the text is not constrained by this limit.

enabled_color contains the color of the edit field when it does not have focus, but only if the edit field is not disabled.

back_color contains the background color for the edit field when it does not have focus, but only if the edit field is not disabled.

disabled_color contains the color of the field if the edit field is disabled.

disabled_back_color contains the background color of the edit field if the edit field is disabled.

active_color contains the foreground color of the edit field if it has the focus. The background color for active edit fields can be set directly in the definition structure. See **XI_FIELD_DEF** for more information.

Return Value

Returns a pointer to the allocated edit field definition.

See Also

XI_OBJ_DEF, **XI_FIELD_DEF**

Example

The following code is from "lstdb.c".

```
for ( num = 0; fielddefs[ num ].width != 0; num++ )
{
  xi_add_field_def( formdef, FIELD_BASE_CID + fielddefs[ num ].type,
                    fielddefs[ num ].v * XI_FU_MULTIPLE,
                    fielddefs[ num ].h * XI_FU_MULTIPLE,
                    fielddefs[ num ].width * 3 / 2 * XI_FU_MULTIPLE,
                    XI_ATR_ENABLED | XI_ATR_VISIBLE
                    | XI_ATR_AUTOSELECT | XI_ATR_AUTOSCROLL
                    | XI_ATR_BORDER,
                    ( fielddefs[ num + 1 ].type == 0 ) ? btn_cid
                    : FIELD_BASE_CID + fielddefs[ num + 1 ].type,
                    fielddefs[ num ].width + 1, COLOR_BLACK,
                    COLOR_WHITE, COLOR_BLACK, COLOR_WHITE,
                    COLOR_BLACK );
}
```

xi_add_form_def — Add Form Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_form_def ( XI_OBJ_DEF *itf, int cid,
                              int tab_cid );
```

Description

This function allocates and initializes a form definition. In addition, it adds it to the list of children for the specified interface definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

A form is necessary to hold edit fields. Edit fields can only be children of forms and edit fields are the only children of forms. The form object does little other than controlling the meta-tab sequence and generating on and off focus events.

itf points to the interface definition. Most likely, the interface definition was previously created via a call to **xi_create_itf_def**. It can also be **NULL** which implies that you will specify an existing interface object in the call to **xi_create** which will add this form, and its edit fields, to that interface.

cid contains the control ID for the form to be created.

tab_cid is the control ID of the form, list, or container that is next in the meta-tabbing sequence. The focus will go to that object when the character defined in the preference **XI_PREF_ITF_TAB_CHAR** is pressed.

Return Value

Returns a pointer to the allocated form definition.

See Also

XI_OBJ_DEF, **XI_FORM_DEF**

Example

The following code is from “lstdb.c”.

```
formdef = xi_add_form_def( itfdef, FORM_CID, FORM_CID );
```

xi_add_group_def — Add Group Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_group_def ( XI_OBJ_DEF *object, int cid,
                               int nbr_cids, int *cid_list );
```

Description

This function allocates and initializes a group definition. In addition, it adds it to the list of children for the specified interface definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

A group is used to associate a number of edit fields or columns so that on and off focus events will be generated whenever focus moves into or out of the group.

itf points to the interface definition. Most likely, the interface definition was previously created via a call to **xi_create_itf_def**. It can also be **NULL** which implies that you will specify an existing interface object in the call to **xi_create** which will add this group to that interface.

cid contains the control ID for the group to be created.

nbr_cids is the number of control ID’s that are in **cid_list**.

cid_list is an array of control ID’s that are in the group. All of the control ID’s in the group must either be edit fields in the same form, or columns in the same list.

Return Value

Returns a pointer to the allocated group definition.

See Also

XI_OBJ_DEF, **XI_GROUP_DEF**

Example

The following code is from “lstdb.c”.

```
{
  int cids[2];

  cids[0] = FIELD_BASE_CID + DB_EMP_MINHRS;
  cids[1] = FIELD_BASE_CID + DB_EMP_MAXHRS;
  xi_add_group_def( itfdef, GROUP_CID, 2, cids );
}
```

xi_add_line_def — Add Line Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_line_def ( XI_OBJ_DEF *itf, int cid,
                              XinPoint *pnt1, XinPoint *pnt2,
                              unsigned long attrib,
                              XinColor fore_color, XinColor
back_color,
                              BOOLEAN well );
```

Description

This function allocates and initializes a line definition. In addition, it adds it to the list of children for the specified interface definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

A line displays in the interface, but cannot have focus. Lines are drawn with a 3D appearance if **XI_PREF_3D_LOOK** is **TRUE**.

itf points to the interface definition. Most likely, the interface definition was previously created via a call to **xi_create_itf_def**. It can also be **NULL** which implies that you will specify an existing interface object in the call to **xi_create** which will add this line to that interface.

cid contains the control ID for the line to be created.

pnt1 is the starting point of the line.

pnt2 is the ending point of the line.

attrib is a bitwise OR’ed combination of **XI_ATR_*** values. The following attributes apply to lines:

```
XI_ATR_VISIBLE
```

fore_color contains the foreground color of the line, if the 3D look is not used.

back_color contains the background color of the line (for XVT/CH only) if the 3D look is not used.

well indicates whether the line is indented, or raised, when using 3D lines.

Return Value

Returns a pointer to the allocated line definition.

See Also

XI_OBJ_DEF, **XI_LINE_DEF**

xi_add_list_def — Add List Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_list_def ( XI_OBJ_DEF *itf, int cid,
                              int v, int h, int height,
                              unsigned long attrib,
                              XinColor enabled_color,
                              XinColor back_color,
                              XinColor disabled_color,
                              XinColor disabled_back_color,
                              XinColor active_color, int tab_cid );
```

Description

This function allocates and initializes a list definition. In addition, it adds it to the list of children for the specified interface definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

The list definition is used to control many aspects of the list. It also contains the column definitions, which will usually be created using **xi_add_column_def**.

itf points to the interface definition. Most likely, the interface definition was previously created via a call to **xi_create_itf_def**. It can also be **NULL** which implies that you will specify an existing interface object in the call to **xi_create** which will add this list, with the defined columns, to that interface.

cid contains the control ID for the list to be created.

v is the vertical position of the top of the list, in form units.

h is the horizontal position of the left edge of the list, in form units.

height is the height of the list, in form units.

attrib is a bitwise OR'ed combination of **XI_ATR_*** values. The following attributes apply to lists:

```
XI_ATR_ENABLED
XI_ATR_VISIBLE
XI_ATR_NAVIGATE
XI_ATR_TABWRAP
```

enabled_color defines the foreground color for text in cells that are not disabled and do not currently have focus. When a cell has focus, the **active_color** is used.

back_color defines the background color for cells that are not disabled and do not currently have focus. When a cell has focus, the **active_back_color** is used.

disabled_color defines the foreground color for text in cells that are disabled.

disabled_back_color defines the background color for cells that are disabled.

active_color defines the foreground color for text in the cell that currently has focus. The background color can be set using the **active_back_color** member of the list definition structure.

tab_cid is the control ID of the form, list, or container that is next in the meta-tabbing sequence. The focus will go to that object when the character defined in the preference **XI_PREF_ITF_TAB_CHAR** is pressed.

Return Value

Returns a pointer to the allocated list definition.

See Also

XI_OBJ_DEF, **XI_LIST_DEF**

Example

The following code is from “lstmem.c”.

```
listdef = xi_add_list_def( itfdef, LIST_CID, 0, 0,
                           8 * XI_FU_MULTIPLE, XI_ATR_ENABLED
                           | XI_ATR_VISIBLE | XI_ATR_TABWRAP,
                           COLOR_BLACK, COLOR_WHITE, COLOR_BLACK,
                           COLOR_WHITE, COLOR_BLACK, LIST_CID );
listdef->v.list->scroll_bar = TRUE;
listdef->v.list->sizable_columns = TRUE;
listdef->v.list->movable_columns = TRUE;
listdef->v.list->fixed_columns = 1;
listdef->v.list->width = 80 * XI_FU_MULTIPLE;
listdef->v.list->select_cells = TRUE;
listdef->v.list->resize_with_window = TRUE;
listdef->v.list->scroll_bar_button = TRUE;
listdef->v.list->drop_and_delete = TRUE;
```

xi_add_rect_def — Add Rectangle Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_rect_def ( XI_OBJ_DEF* itf, int cid,
                              XinRect* rct, unsigned long attrib,
                              XinColorfore_color,
                              XinColor back_color );
```

Description

This function allocates and initializes a rectangle definition. In addition, it adds it to the list of children for the specified interface definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

A rectangle displays in the interface, but cannot have focus. Rectangles are drawn with a 3D appearance if **XI_PREF_3D_LOOK** is **TRUE**.

It is very important to add your rectangles to your XI interface before adding other types of objects. XI draws the objects in the order added. If you add rectangles after you add other objects such as the form containing edit fields, then the rectangle will be drawn after the edit fields, and obliterate them. The most common mistake is to add an XI form, add a rectangle, then add the edit fields to the form. In this case, because of the hierarchy of controls, the form, and thus the edit fields in it, get drawn before the rectangle.

itf points to the interface definition. Most likely, the interface definition was previously created via a call to **xi_create_itf_def**. It can also be **NULL** which implies that you will specify an existing interface object in the call to **xi_create** which will add this rectangle to that interface.

cid contains the control ID for the rectangle to be created.

rct contains the coordinates of the rectangle, in form units.

attrib is a bitwise OR'ed combination of **XI_ATR_*** values. The following attributes apply to rectangles:

```
XI_ATR_VISIBLE
```

fore_color contains the foreground color of the rectangle, if the 3D look is not used.

back_color contains the background color of the rectangle (for XVT/CH only) if the 3D look is not used.

Return Value

Returns a pointer to the allocated rectangle definition.

See Also

XI_OBJ_DEF, **XI_RECT_DEF**

Example

The following code is from “lstlink.c”.

```
{
  XinRect      rct;
  XI_OBJ_DEF* cntrdef;
  XI_OBJ_DEF* btndef;

  rct.top = 3 * XI_FU_MULTIPLE;
  rct.bottom = rct.top + 8 * XI_FU_MULTIPLE;
  rct.left = XI_FU_MULTIPLE;
  rct.right = 43 * XI_FU_MULTIPLE;
  xi_add_rect_def( itfdef, RECT_CID, &rct, XI_ATR_VISIBLE,
                   COLOR_BLACK, COLOR_WHITE );
  ...
}
```

xi_add_text_def — Add Text Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_add_text_def ( XI_OBJ_DEF* itf, int cid,
                              XinRect* rct, unsigned long attrib,
                              char* text );
```

Description

This function allocates and initializes a static text definition. In addition, it adds it to the list of children for the specified interface definition, if any. Object definitions are used to create objects by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

The text displays in the interface, but cannot have focus. Since it cannot have focus and cannot be edited, it is usually referred to as “static” text. However, you can update the displayed text by calling **xi_set_text** on a created text object. Text is most commonly used as labels for edit fields.

itf points to the interface definition. Most likely, the interface definition was previously created via a call to **xi_create_itf_def**. It can also be **NULL** which implies that you will specify an existing interface object in the call to **xi_create** which will add this text to that interface.

cid contains the control ID for the static text to be created.

rct is the bounding for the text, in form units.

attrib is a bitwise OR’ed combination of **XI_ATR_*** values. The following attributes apply to text:

```
XI_ATR_ENABLED
XI_ATR_RJUST
XI_ATR_VISIBLE
```

text contains the initial text to be displayed. The text may be changed, after the object is created, by calling **xi_set_text**.

Return Value

Returns a pointer to the allocated static text definition.

See Also

XI_OBJ_DEF, **XI_TEXT_DEF**

Example

The following code is from "lstdb.c".

```
for ( num = 0; textdefs[ num ].text != NULL; num++ )
{
  XinRect rct;

  rct.top = textdefs[ num ].v * XI_FU_MULTIPLE;
  rct.left = textdefs[ num ].h * XI_FU_MULTIPLE;
  rct.bottom = rct.top + XI_FU_MULTIPLE;
  rct.right = rct.left + strlen( textdefs[ num ].text )
              * 3 / 2 * XI_FU_MULTIPLE;
  xi_add_text_def( itfdef, TEXT_BASE_CID + num, &rct,
                   XI_ATR_VISIBLE | XI_ATR_ENABLED,
                   textdefs[ num ].text );
}
```

xi_bitmap_create

Summary

```
XI_BITMAP* xi_bitmap_create ( char* filename, XI_BITMAP_MODE mode);
```

Description

This creates a pointer to a bitmap.

filename is the full pathname for the bitmap, the bitmap cannot be placed in resource, so it must be supplied as an external file.

mode is the mode the bitmap will be drawn with.

Return Value

Returns a pointer to the bitmap.

See Also

XI_BITMAP, **XI_BITMAP_MODE**

Example

The following code is from “lstsort.c”.

```
btn_def->v.btn->up_bitmap = xi_bitmap_create("okup.bmp",
XI_BITMAP_NORMAL );
  btn_def->v.btn->down_bitmap = xi_bitmap_create("okdn.bmp",
XI_BITMAP_NORMAL );
```

xi_bitmap_destroy

Summary

```
void xi_bitmap_destroy ( XI_BITMAP* bitmap );
```

Description

This destroys all memory allocated for a bitmap. It is not necessary to destroy a bitmap that has been assigned to a XI object.

bitmap is the pointer to a bitmap created with **xi_bitmap_create.**

See Also

xi_bitmap_create

xi_bitmap_draw

Summary

```
void xi_bitmap_draw ( XI_BITMAP* bitmap, XinWindow win,
                      XinRect* rct, XinRect* clip_rct,
                      BOOLEAN in_paint_event );
```

Description

This function draws a bitmap. It should be called only for bitmaps which have not been assigned to XI objects. XI handles the drawing of bitmaps for those objects.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

win is window into which the bitmap will be drawn.

rct is the bounding for the bitmap, in pixels.

clip_rct is the clipping rectangle for the bitmap, in pixels

in_paint_event should be FALSE unless the draw is being done in a update event.

See Also

XI_BITMAP, **XI_BITMAP_MODE xi_bitmap_create xi_bitmap_destroy**

Example

The following code is from "main.c".

```
case E_UPDATE:
    {
      RCT rect;
      xvt_vobj_get_client_rect( win, &rect );
      if ( task_bitmap != NULL )
        xi_bitmap_draw( task_bitmap, (XinWindow)win,
             (XinRect *)&rect, NULL, TRUE );
```

xi_bitmap_size_get

Summary

```
void xi_bitmap_size_get ( XI_BITMAP* bitmap, short* width,
                          short* height ) );
```

Description

This function determines the width and height of a bitmap.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

width is a pointer to a short which will be filled with the width of the bitmap in pixel units.

height is a pointer to a short which will be filled with the height of the bitmap in pixel units.

xi_bitmap_draw_all_on_resize

Summary

```
BOOLEAN xi_bitmap_draw_all_on_resize ( XI_BITMAP* bitmap );
```

Description

This function determines if the entire bitmap needs to be redrawn in response to a size event. It should be called only for bitmaps that have not been assigned to XI objects.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

See Also

XI_BITMAP, **XI_BITMAP_MODE xi_bitmap_create xi_bitmap_destroy**

Example

The following code is from “main.c”.

```
case E_SIZE:
      if ( xi_bitmap_draw_all_on_resize( task_bitmap ) )
        xvt_dwin_invalidate_rect( win, NULL );
      break;
```

xi_bitmap_background_get

Summary

```
#define xi_bitmap_background_get( bitmap ) ((bitmap)->background)
```

Description

This macro returns the background color associated with a bitmap which is drawn in normal mode.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_hcenter_get

Summary

```
#define xi_bitmap_hcenter_get( bitmap ) ((bitmap)->hcenter)
```

Description

This macro returns the flag which determines if a bitmap, drawn in normal mode, is horizonally centered.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_mode_get

Summary

```
#define xi_bitmap_mode_get( bitmap ) ((bitmap)->mode)
```

Description

This macro returns the mode the bitmap is set to draw in.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_offset_x_get

Summary

```
#define xi_bitmap_offset_x_get( bitmap ) ((bitmap)->x_offset)
```

Description

This macro returns the x offset of a bitmap which is drawn in normal mode.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_offset_y_get

Summary

```
#define xi_bitmap_offset_y_get( bitmap ) ((bitmap)->y_offset)
```

Description

This macro returns the y offset of a bitmap which is drawn in normal mode.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_vcenter_get

Summary

```
#define xi_bitmap_vcenter_get( bitmap ) ((bitmap)->vcenter)
```

Description

This macro returns the flag which determines if a bitmap, drawn in normal mode, is vertically centered.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_background_set

Summary

```
#define xi_bitmap_background_set( bitmap, color )
   ((bitmap)->background = (color) )
```

Description

This macro sets the background color associated with a bitmap which is drawn in normal mode.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

color is the background color.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_hcenter_set

Summary

```
#define xi_bitmap_hcenter_set( bitmap, flag )
   ((bitmap)->hcenter = (flag) )
```

Description

This macro sets the flag which determines if a bitmap, drawn in normal mode, is centered horizontally.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

flag is a BOOLEAN indicating if the bitmap is centered horizontally.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_mode_set

Summary

```
#define xi_bitmap_mode_set( bitmap, m )
    ((bitmap)->mode = (m) )
```

Description

This macrosets the mode the bitmap will be drawn in.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

m is the **XI_BITMAP_MODE**.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_offset_x_set

Summary

```
#define xi_bitmap_offset_x_set( bitmap, x )
   ((bitmap)->x_offset = (x) )
```

Description

This macro sets the x offset of a bitmap which is drawn in normal mode.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

x is the offset.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_offset_y_set

Summary

```
#define xi_bitmap_offset_y_set( bitmap, y )
   ((bitmap)->y_offset = (y) )
```

Description

This macro sets the y offset of a bitmap which is drawn in normal mode.

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

y is the offset.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_bitmap_vcenter_set

Summary

```
#define xi_bitmap_vcenter_set( bitmap, flag )
   ((bitmap)->vcenter = (flag) )
```

Description

This macrosets the flag which determines if a bitmap, drawn in normal mode, is vertically centered..

bitmap is a pointer to the bitmap, returned from **xi_bitmap_create**.

flag is a BOOLEAN which determines if the bitmap is vertically centered.

See Also

XI_BITMAP, **XI_BITMAP_MODE, xi_bitmap_create**

xi_button_calc_height_font_id

Summary

```
int xi_button_calc_height_font_id( XVT_FNTID font );
```

Description

This function determines the required height of a button, in pixels, for the specified font. The result can be used as the button height for “grid” buttons.

font is the XVT font used for the calculation.

Return Value

Returns the pixel height for a button containing **font**.

xi_button_calc_pixel_height

Summary

```
int xi_button_calc_pixel_height( int height );
```

Description

Returns the height of a button, in pixels, that has a “drawable” height. This can be used to set the button height for “grid” buttons that is large enough to hold a specific image, e.g. an icon.

height is the number of pixels that will be available for drawing or icon images on the button.

Return Value

Returns the pixel height for a button.

xi_button_calc_pixel_width

Summary

```
int xi_button_calc_pixel_width( int width );
```

Description

Returns the width of a button, in pixels, that has the specified “drawable” width. This can be used to set the button width for “grid” buttons that is large enough to hold a specific image, e.g. an icon.

width is the number of pixels that will be available for drawing or icon images on the button.

Return Value

Returns the pixel width for a button.

xi_button_def_get_width

Summary

```
int xi_button_def_get_width( XI_OBJ_DEF* btn_def );
```

Description

Returns the width of a button definition, in pixels. This can be used as button width for “grid” buttons. For example, you could find the largest width for all the buttons in the grid and set the button width to that maximum.

btn_def is the button definition to get the width for.

Return Value

Returns the pixel width for **btn_def**.

xi_button_set_default

Summary

```
BOOLEAN xi_button_set_default( XI_OBJ* obj, BOOLEAN set );
```

Description

This function is used to change the default button for an interface.

If **obj** is a button, **set** can be either **TRUE** or **FALSE**. If **TRUE**, the button becomes the default button and any other button in the interface that was the default will no longer be the default. If **FALSE**, the button is set to not be the the default button.

If **obj** is a button container, **set** must be **FALSE**. If the default button for the interface is in the container, it will be set to not be the default.

If **obj** is an interface, **set** must be **FALSE**. If the interface has a default button, the button will be set to not be the default button.

Return Value

Returns the **FALSE** if the above constraints were violated in any way.

xi_cell_request Generate Cell Request Events

Summary

```
void xi_cell_request ( XI_OBJ *xi_obj );
```

Description

This function forces **XIE_CELL_REQUEST** events for every cell in **xi_obj**. However, if the cells are not currently visible and **XI_PREF_OPTIMIZE_CELL_REQUESTS** is **TRUE**, the cell requests will not occur until the cells become visible.

If **xi_obj** is a cell, a cell request event is sent for just that cell.

If **xi_obj** is a column, a cell request event is generated for every cell in the column.

If **xi_obj** is a row, then a cell request event is generated for every cell in the row.

If **xi_obj** is a list, then a cell request event is generated for every cell in the list.

Example

The following code is from “lstlink.c”.

```
static void update_numbers( XI_OBJ* list )
{
  XI_OBJ* column = xi_get_obj( list, COL_BASE_CID + LINK_NUM );

  if ( column != NULL )
    xi_cell_request( column );
}
```

xi_check — Check a Button

Summary

```
void xi_check( XI_OBJ *xi_obj, BOOLEAN check );
```

Description

This function checks or unchecks a radio button, a check box, or a tab button. In general, you should not uncheck a radio button, since radio buttons automatically uncheck when another radio button is checked. (Note that radio buttons must be in a container to check and uncheck properly.)

xi_obj points to the object that you want to check.

check is the desired state of the button.

Example

The following code is from “lstlink.c”.

```
void form_process_button( XI_OBJ* itf, XI_OBJ* button )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( button->cid )
  {
    ...
    case SECTION_ONE_CID:
    case SECTION_TWO_CID:
    {
      SECTION_INFO* info = (SECTION_INFO*)xi_get_app_data( button );

      if ( form_info->cur_section != info )
      {
        change_section( form_info->cur_section, FALSE );
        change_section( info, TRUE );
        form_info->cur_section = info;
      }
      xi_check( button, TRUE );
      break;
    }
    ...
  }
}
```

xi_clean_up Clean up memory

Summary

```
void xi_clean_up();
```

Description

Allows XI to clean up stray memory on exit. This should be placed in the E_DESTROY of the Task window event handler.

xi_column_set_pixel_width

Summary

```
void xi_column_set_pixel_width( XI_OBJ *column, int width );
```

Description

Sets the width of a column in pixels.

column is the column object whose width is to be changed.

width is the new width in pixels.

See Also

xi_set_column_width

xi_container_def_get_btn_width

Summary

```
int xi_container_def_get_btn_width( XI_OBJ_DEF* cntr_def );
```

Description

This function determines the minimum width, in pixels, that is large enough that all of the buttons in a button container definition can be displayed correctly. This value can be used as the button width for “grid” buttons.

cntr_def is the button container definition to use for the calculation.

Return Value

Returns the minimum button width, in pixels.

xi_container_def_get_height

Summary

```
int xi_container_def_get_height( XI_OBJ_DEF* cntr_def );
```

Description

This function determines the minimum height, in pixels, that is valid for a button container definition. This value can then be used to specify a pixel rectangle for the container.

cntr_def is the button container definition to use for the calculation.

Return Value

Returns the height of the container, in pixels.

xi_container_def_get_width

Summary

```
int xi_container_def_get_width( XI_OBJ_DEF* cntr_def );
```

Description

This function determines the minimum width, in pixels, that is valid for a button container definition. This value can then be used to specify a pixel rectangle for the container.

cntr_def is the button container definition to use for the calculation.

Return Value

Returns the width of the container, in pixels.

xi_container_reorient Change the Orientation of a Button Container

Summary

```
void xi_container_reorient( XI_OBJ *cnt_obj,
                            XI_CONTAINER_DEF *cnt_def );
```

Description

This function changes the arrangement of the buttons in an existing container. With this function, your application can change the bounding rectangle of a container, causing the placement of the buttons within the container to change. In addition, an application can change the orientation and packing.

cnt_obj is the container to be reoriented.

cnt_def is a pointer to an **XI_CONTAINER_DEF** structure that defines the new information about the container.

See Also

XI_CONTAINER_DEF

xi_create Create an XI Object

Summary

```
XI_OBJ* xi_create( XI_OBJ *parent, XI_OBJ_DEF* xi_obj_def );
```

Description

xi_create creates objects from their definitions. The definitions are usually created using **xi_create_itf_def** and the **xi_add_*_def** convenience functions.

parent is the object that the definition or definitions will be added to. If **xi_obj_def** is an interface, this value should be **XI_NULL_OBJ**. When an interface is created, it will also create an XVT window for the controls in that interface (unless you have already created the XVT window and set the **win** member of the interface definition). If **xi_obj_def** is any other type, the **parent** object must be the appropriate type. For example, if **xi_obj_def** is a column definition, **parent** must be a list object. The **parent** object should never be a row or cell object.

xi_obj_def is the object definition. That object and all of its children and their children, if any, will be created. After this function call, the allocation of the object definitions can be freed, usually by using **xi_tree_free** or **xi_def_free**.

You should not need to save the pointer to any object for later use, except possibly the interface itself. Object pointers are passed into XI events. You can use **xi_get_obj** to get an object pointer within an interface using its control ID. You can use **xi_get_itf** to get an interface pointer from an XVT **WINDOW** handle. The list of children for an object is available using the **xi_get_member_list** function.

Tree memory allocation is used for all data in an interface.

Return Value

Returns a pointer to the created object.

Example

The following code is from “lstdb.c”.

```
xi_create( NULL, itfdef );
xi_dequeue();
xi_tree_free( itfdef );
```

xi_create_itf_def Create Interface Definition Convenience Function

Summary

```
XI_OBJ_DEF* xi_create_itf_def( int cid, XI_EVENT_HANDLER xi_eh,
                               XinRect *rctp, char *title,
                               long app_data );
```

Description

This function allocates and initializes an interface definition. The interface object is created from the definition by calling **xi_create**. The memory is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

The interface represents a "window" and all of its controls. The definitions for those controls are added to the interface or to children of the interface. For example, a list definition is a direct child of the interface while column definitions are children of the list definition. The call to **xi_create** will translate all of the definitions into actual objects.

cid contains the control ID for the interface to be created.

xi_eh is a pointer to the application program's event handler function for this interface.

rctp is used as the position and size for the XVT window that is created, in pixels. If this is set to **NULL**, the window will be sized according to the controls in the interface and positioned reasonably. If you set the rectangle so that the bottom and top are the same and so that the left and right are the same, the window will be created at the specified position and sized according to the controls in the interface.

title contains the title for the XVT window. This can be changed after creation by calling **xi_set_text** on the interface object.

app_data will be stored in the interface object after creation. It can be retrieved by calling **xi_get_app_data** on that interface object.

Return Value

Returns a pointer to the allocated interface definition.

Example

The following code is from "lstdb.c".

```
itfdef = xi_create_itf_def( ITF_CID, (XI_EVENT_HANDLER)list_eh,
NULL,
                            "Employee List", 0L );
itfdef->v.itf->ctl_size = TRUE;
itfdef->v.itf->menu_bar_rid = MENU_BAR_RID;
itfdef->v.itf->automatic_back_color = TRUE;
itfdef->v.itf->edit_menu = TRUE;
itfdef->v.itf->whitespace_right = 0;
itfdef->v.itf->whitespace_bottom = 0;
itfdef->v.itf->use_whitespace = TRUE;
```

xi_def_free — Free Definition Memory

Summary

```
void xi_def_free( XI_OBJ_DEF* obj_def );
```

Description

This function frees any bitmaps and fonts that XI has associated with this definition structure and then calls **xi_tree_free**.

obj_def contains the definition to be freed.

xi_delete — Delete an XI Object

Summary

```
void xi_delete( XI_OBJ *xi_obj );
```

Description

xi_obj is a pointer to the object to be deleted. This object can be any type except for cells and rows. Rows can be deleted using **xi_delete_row**. Cells can only be deleted by deleting a column or row.

Deleting an interface object will also close the window that contains that interface. Any tree memory parented to the interface object will also be freed, after the final **XIE_CLEANUP** event. Calling **xi_delete** on an interface during an event for that interface will delay the actual destruction of the window until you return from the event handler function. This is done so that all the objects in that window remain valid for any other processing you may do in that event.

For objects that are not the interface, you can use this function in combination with **xi_create** to make large changes in the appearance of the interface while the window remains open. You can use **xi_get_def** on most objects, delete the object using this function, change the values in the definition structure, and then create the updated object using **xi_create** with the appropriate parent.

This function is called internally if you do not refuse an **XIE_CLOSE** event.

Example

The following code is from "lstdb.c".

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_BUTTON:
      switch ( xiev->v.xi_obj->cid )
      {
        ...
        case CANCEL_BTN_CID:
          if ( form_info->changed && xvt_ask( "No", "Yes", NULL,
    "You have made changes. Are you sure you want to cancel?" )
              != RESP_2 )
            break;
          xi_delete( itf );
          break;
      }
      break;
    ...
  }
}
```

xi_delete_row — Delete a Row in a List

Summary

```
BOOLEAN xi_delete_row ( XI_OBJ *xi_obj );
```

Description

This function causes the specified row to be removed by scrolling all the rows below it upward. You should be sure that you can respond correctly to any **XIE_GET_*** events before calling this function. For example, if your internal structure is based on a database file, you should delete the record from the database and then call **xi_delete_row**.

This function is preferred to calling **xi_scroll** or **xi_scroll_rec** because it will optimize updating the displayed data and minimizing the number of record and cell requests.

Before deleting the row, **xi_delete_row** attempts to move the focus to the interface.

xi_obj is the row object to be deleted.

Return Value

Returns **TRUE** if the row was deleted. **xi_delete_row** attempts to move the focus to the interface. If any focus change events are refused, then **xi_delete_row** will return **FALSE** and the row will not be deleted.

Example

The following code is from "lstlink.c".

```
static void process_button( XI_OBJ* itf, XI_OBJ* button )
{
  LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

  ...
  switch ( button->cid )
  {
    ...
    case DELETE_CUR_CID:
    {
      XI_OBJ* obj = xi_get_focus( itf );

      if ( obj->type == XIT_CELL )
      {
        XI_OBJ row;

        link_delete( row_to_record( obj->parent, obj-
>v.cell.row ) );
        XI_MAKE_ROW( &row, obj->parent, obj->v.cell.row );
        xi_delete_row( &row );
        update_numbers( obj->parent );
      }
      break;
    }
    ...
  }
}
```

xi_dequeue — Dequeue All Events After Creating an Interface

Summary

```
void xi_dequeue(void);
```

Description

This function causes any initial events for the interface to be sent to the event handler function. Most importantly, it causes the **XIE_INIT** event to be generated. It should be called after **xi_create**, but this is only necessary on XVT/XM platforms.

This functionality is separated from **xi_create** so that you can initialize the application data for objects within the interface before you start getting events for them.

Example

The following code is from "lstdb.c".

```
xi_create( NULL, itfdef );
xi_dequeue();
xi_tree_free( itfdef );
```

xi_event Pass an XVT Event to XI

Summary

```
void xi_event( WINDOW win, EVENT *ep );
```

Description

This function must be called for every XVT event in a window containing an XI interface. If it isn't the controls may not function correctly. This function is the XVT event handler for any window that is created by XI. If you create the window yourself, you must specify this function as the event handler or you must call it for every event in your own event handler. Since XI passes all XVT events to your XI event handler function, it should not be necessary to write an XVT event handler for an XI window.

win is the XVT window passed to the XVT event handler.

ep is a pointer to the XVT event.

Example

The following code, from "lstdb.c", demonstrates using **xi_event** as the XVT event handler when creating the window.

```
{
  RCT r;

  xi_get_def_rect( itfdef, &r );
  xvt_rect_offset( &r, (short)xi_get_pref( XI_PREF_ITF_MIN_LEFT ),
                   (short)xi_get_pref( XI_PREF_ITF_MIN_TOP ) );
  itfdef->v.itf->win = xvt_win_create( W_DOC, &r, "Employee List",
                                       MENU_BAR_RID, TASK_WIN,
                                       WSF_SIZE | WSF_CLOSE
                                       | WSF_ICONIZABLE, EM_ALL,
                                       (EVENT_HANDLER)xi_event,
0L );
}
```

xi_event_debug — Convert an XI Event to a Displayable String

Summary

```
void xi_event_debug( char *tag, XI_EVENT *xiev, char *s,
                     int len );
```

Description

This event makes a displayable string out of an **XI_EVENT** structure. When debugging, it is sometimes helpful to see all of the events that your application has received. After calling this function, you can print the resulting string to a file or use **xvt_debug** or **xvt_debug_printf** to log the events. Often, it is useful to filter out **E_MOUSE_MOVE** XVT events for both **XIE_XVT_EVENT** and **XIE_XVT_POST_EVENT** events.

tag points to a string that is placed in the displayable string at its beginning.

xiev points to the XI event structure that you want to convert to a displayable string.

s points to a buffer which the displayable string will be copied to. Usually, 200 characters is enough for the size of the buffer.

len is the length of buffer **s**. Usually this value will be **sizeof(s)**.

Example

The following code is from “lstdb.c”.

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

#ifdef DEBUG_EVENTS
  if ( xiev->type != XIE_XVT_EVENT
       && xiev->type != XIE_XVT_POST_EVENT )
  {
    char s[200];

    xi_event_debug( "lstdb-form_eh", xiev, s, sizeof(s) );
    xvt_debug_printf( s );
  }
#endif

  ...
}
```

xi_field_calc_height_font_id

Summary

```
int xi_field_calc_height_font_id( XVT_FNTID font );
```

Description

Returns the minimum height for an edit field, in pixels, based on a font. This value can be used to set the pixel rectangle for an edit field.

font is the XVT font used for the calculation.

Return Value

Returns the height in pixels.

xi_field_calc_width_font_id

Summary

```
int xi_field_calc_width_font_id( XVT_FNTID font, char* string );
```

Description

Returns the minimum width for an edit field, in pixels, that is wide enough to show the entire specified string.

font is the XVT font that will be used for the string.

string is the text that will be displayed in the edit field.

Return Value

Returns the width in pixels.

xi_fu_to_pu Convert Form Units to Pixel Units

Summary

```
void xi_fu_to_pu( XI_OBJ *itf, PNT *pnts, int nbr_pnts );
```

Description

This function converts the coordinates in an array of points from form units to pixels. Note that because a rectangle can be cast into an array of points with two elements, it is possible to call this function to convert coordinates in a rectangle from form units to pixels.

itf is used to get the form unit metrics, based on the font for the interface, for the conversion.

pnts points to an array of points to be converted. If you have only one point to convert, you should pass the address of that point.

nbr_pnts is the number of points to be converted in the **pnts** array.

See Also

xi_pu_to_fu

xi_get_app_data Get Application Data

Summary

```
long xi_get_app_data( XI_OBJ *xi_obj );
```

Description

This function returns the application data stored in an XI object. You can set this data for an interface by specifying it when creating the interface definition. You can also set this data by calling **xi_set_app_data** for any XI object, except rows and cells.

xi_obj is the object for which you want to get the application data.

See Also

xi_set_app_data

Example

The following code is from "lstdb.c".

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  ...
}
```

xi_get_app_data2 Internal Function

Summary

```
long xi_get_app_data2( XI_OBJ *xi_obj );
```

Description

This function is for internal use only.

xi_get_attrib Get Object Attributes

Summary

```
unsigned long xi_get_attrib( XI_OBJ *xi_obj );
```

Description

This function retrieves the XI attributes that are set for an object. The attributes are a bitwise OR'ed combination of the **XI_ATR_*** values. The attributes for an object can be set in the definition, or by calling **xi_set_attrib** on the object.

xi_obj is the object for which you want to get attributes.

Return Value

Returns the attributes of the object.

See Also

xi_set_attrib

Example

The following code, from “lstlink.c”, demonstrates making controls visible or invisible.

```
static void change_section( SECTION_INFO* section, BOOLEAN flag )
{
  int      num;
  XI_OBJ** obj;

  for ( num = 0, obj = section->objs; num < section->count;
        num++, obj++ )
    if ( flag )
      xi_set_attrib( *obj, xi_get_attrib( *obj ) | XI_ATR_VISIBLE );
    else
      xi_set_attrib( *obj, xi_get_attrib( *obj ) &
~XI_ATR_VISIBLE );
}
```

xi_get_cell_selection Get a List of Selected Cells

Summary

```
XI_CELL_SPEC* xi_get_cell_selection( XI_OBJ *list,
                                     int *nbr_cells );
```

Description

This function returns the entire set of selected cells. It would most often be used after the user has selected one or more ranges of cells and wishes to perform some action on the selected cells.

list is the list object that you want to get the cell selection list for.

nbr_cells is a pointer to an integer value that will be set to the number of cells that were selected.

Return Value

Returns an array of cell specifications, one for each of the selected cells. The total size of this array is stored in **nbr_cells**.

Example

The following code is from “lstmem.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_SELECT:
      switch ( xiev->v.select.xi_obj->type )
      {
        ...
        case XIT_LIST:
        {
          XI_CELL_SPEC* cells;
          int count;
          int num;
          XI_OBJ* list = xiev->v.select.xi_obj;

          cells = xi_get_cell_selection( list, &count );
          select_clear( list->itf, FALSE, FALSE );
          for ( num = 0; num < count; num++, cells++ )
            mem_select_cell( row_to_record( list, cells->row ),
                             column_to_code( list, cells->column ),
                             xiev->v.select.selected );
          break;
        }
      }
      break;
    ...
  }
}
```

xi_get_def — Get an Object Definition from an Object

Summary

```
XI_OBJ_DEF* xi_get_def( XI_OBJ *xi_obj );
```

Description

This function returns an object definition that is initialized to match the current state of the object specified by **xi_obj**. This definition can be modified and then used to create or recreate the control by calling **xi_create**. NOTE: In order to maintain compatibility with XVT R3, the **font** member of many objects is set. If you want to set the **font_id** for an object, you should set the **font** member to **NULL** before calling **xi_create**.

xi_obj can be a list, column, edit field, static text, line, rectangle, or button control. If the object is a list, the current columns of the list will be children in the definition also. Notice that you can use this function to get information about the current appearance of the list in order to save user changes to column order and width.

Return Value

Returns an **XI_OBJ_DEF** for the object pointed to by **xi_obj**. This definition is allocated using tree memory and can be freed by calling **xi_tree_free** or **xi_def_free**.

Example

The following code, from "lstmem.c", saves the current column definition for possible use later.

```
static BOOLEAN do_col_delete( LIST_INFO* list_info, XI_EVENT* xiev )
{
  int                 count;
  struct xit_column*  column = &xiev->v.column;
  XI_OBJ**            members = xi_get_member_list( column->list,
                                &count );
  XI_OBJ_DEF*         column_def;

  column_def = xi_get_def( members[column->col_nbr] );
  add_column_def( &list_info->deleted_column_list, column_def );
  return TRUE;
}
```

xi_get_def_rect Get Definition Rectangle

Summary

```
XinRect* xi_get_def_rect( XI_OBJ_DEF *xi_obj_def, XinRect *rctp );
```

Description

This function determines the bounding rectangle, in pixels, from an object definition. Use **xi_get_rect** for created objects.

Most commonly, this function is called on the interface definition to get the bounding rectangle for all objects. This rectangle can then be used to create an XVT window yourself which will be set as the **win** argument in the interface definition.

This function can also be used to get the pixel positions of specific XI controls so that XVT controls can also be placed in the window without overlapping the XI controls. For example, you could get the rectangle for the rightmost edit field and then place XVT buttons to the right of that.

xi_obj_def is the object definition for which you want to get a bounding rectangle.

rctp is a pointer to a rectangle which will be filled in with the bounding rectangle, in pixel units. This pointer is also returned by **xi_get_def_rect** so that you can call this function within a call to another function that takes a pointer to a rectangle as an argument.

Return Value

Returns the value of **rctp**. The rectangle pointed to by **rctp** is filled in with the bounding rectangle of the object definition, in pixels.

Example

The following code is from “main.c”.

```
void center_interface( XI_OBJ_DEF* itfdef )
{
  RCT  r;
  int  width, height;
  XI_RCT* itf_rct = itfdef->v.itf->rctp;

  xi_get_def_rect( itfdef, (XI_RCT *)&r );
  width = r.right - r.left;
  height = r.bottom - r.top;
  xvt_vobj_get_client_rect( TASK_WIN, &r );
  itf_rct->top = ( r.bottom - height ) / 2;
  if ( itf_rct->top < xi_get_pref( XI_PREF_ITF_MIN_TOP ) )
    itf_rct->top = (int)xi_get_pref( XI_PREF_ITF_MIN_TOP );
  itf_rct->left = ( r.right - width ) / 2;
  if ( itf_rct->left < xi_get_pref( XI_PREF_ITF_MIN_LEFT ) )
    itf_rct->left = (int)xi_get_pref( XI_PREF_ITF_MIN_LEFT );
  itf_rct->bottom = itf_rct->top;
  itf_rct->right = itf_rct->left;
  if ( itfdef->v.itf->modal )
    xvt_vobj_translate_points( TASK_WIN, SCREEN_WIN,
                               (PNT*)itf_rct, 2 );
}
```

xi_get_fixed_columns

Summary

```
int xi_get_fixed_columns( XI_OBJ* list );
```

Description

This function is used to get the current number of fixed columns for a horizontally scrolling list.

list is the list object to get the fixed columns for.

Return Value

Returns the current number of fixed columns for **list**.

xi_get_focus Get the Object that has Focus

Summary

```
XI_OBJ* xi_get_focus( XI_OBJ *itf );
```

Description

This function returns the object that currently has focus in a particular interface. Although only one object, in all interfaces, can actually have focus at any time, each interface keeps track of which object will have focus when its window has focus. Only cells, edit fields and buttons can have focus. Buttons cannot have focus on the Macintosh.

itf points to the interface for which you want to know the focus object. If this value is **NULL**, the object with focus in the window that has focus will be returned.

Return Value

Returns a pointer to the object with the focus or **XI_NULL_OBJ** if no object has the focus.

Example

The following code is from “lstmem.c”.

```
static void do_mem_menu( XI_OBJ* itf, MENU_TAG cmd )
{
  XI_OBJ* obj;

  switch ( cmd )
  {
    ...
    case M_SELECT_ROW:
      obj = xi_get_focus( itf );
      if ( obj->type == XIT_CELL && xi_move_focus( itf ) )
      {
        XI_OBJ row;

        XI_MAKE_ROW( &row, obj->parent, obj->v.cell.row );
        mem_select_row( row_to_record( obj->parent,
                        obj->v.cell.row ), TRUE );
        xi_set_attrib( &row, xi_get_attrib( &row )
                       | XI_ATR_SELECTED );
      }
      break;
    ...
  }
}
```

xi_get_itf — Get the Interface From a WINDOW

Summary

```
XI_OBJ* xi_get_itf( WINDOW win );
```

Description

This function allows you to retrieve the interface object for a window.

win is the XVT window handle.

Return Value

Returns a pointer to the interface object for the window. If the window is not a valid handle, or does not contain an XI interface, then **NULL** is returned.

xi_get_list_info — Get Record Handle Information From a List

Summary

```
long* xi_get_list_info( XI_OBJ *list, int *nbr_recs );
```

Description

This function is used to get the current array of record handles for the list. These record handles are set by either the **XIE_REC_ALLOCATE** event or the **XIE_GET_*** events. This array can be used to translate row numbers from cell and row objects into the corresponding handle. The row number is simply used as an index into the array returned from this function.

The array of record handles will usually be just the visible rows. However, the **get_all_records** option, if **TRUE**, will cause the record handle list to hold all of the records for the list. Future enhancements may also change the number of records actually in the array, but the row number translation will always be correct.

list is the list object for which you want to get list information. You most often would get the list object by calling **xi_get_obj**.

nbr_rows points to an integer that will be filled with the number of that are currently in the record handle array.

Return Value

Returns an array of longs that are the current record handles. This array will probably change whenever the list scrolls. If you need to keep the resulting array information around for long, you should copy it.

See Also

xi_get_visible_rows

Example

The following code is from “lstmem.c”.

```
static REC_INFO* row_to_record( XI_OBJ* list, int row_num )
{
  int count;

  long* handles = xi_get_list_info( list, &count );
  return ((LIST_INFO*)xi_get_app_data( list->itf ))->records
         + (int)handles[ row_num ];
}
```

xi_get_member_list Get the Contained Objects for an Object

Summary

```
XI_OBJ** xi_get_member_list( XI_OBJ *xi_obj, int *nbr_members );
```

Description

This function is used to get the list of members for an object. For example, if you call it for a list object, the array will contain all of the column objects for the list. You can then use the result to find out what order the columns are on the list, for example. Also, you may need to do some processing on all of the members, but do not have easy access to the control ID’s for all of those objects.

A common use of this function is to get the column object, given a cell object. To get the column object, call **xi_get_member_list** for the list object, and index into the returned array with the column number.

xi_obj points to the object for which you want to get all members. This will usually be an object that contains other objects, like interfaces, lists, forms and buttons containers. This function also works on groups, but its objects will also be members of some other object in the interface. This means that if you write a recursive processing function that makes this call, you may want to handle groups as a special case.

nbr_members points to an integer that will be filled with the number of contained members. **nbr_members** is also the number of objects in the array returned by this function. For controls like edit fields that have no members, this integer will be set to zero.

Return Value

Returns an array of object pointers. The object pointers can be used in various XI functions, but they should not be freed or changed directly.

Example

The following code is from "lstdb.c".

```
static DB_EMP_FIELD column_to_field( XI_OBJ* list, int col_num )
{
  int count;

  XI_OBJ** members = xi_get_member_list( list, &count );
  return members[ col_num ]->cid - COL_BASE_CID;
}
```

xi_get_obj Get an Object Given the Control ID

Summary

```
XI_OBJ* xi_get_obj( XI_OBJ *xi_obj, int cid );
```

Description

This is one of the most commonly called functions in XI. It finds an object using its control ID.

xi_obj is the object that contains the object to which you want a pointer. Almost always, this will be an interface object. Objects that are not direct children of the interface will be found by this function. For example, you can find a column that is a member of a list which is a member of the interface. You can use other containing objects (lists, forms, button containers) as this argument. In that case, the control will only be found if it is a member of that containing object.

cid is the control ID of the object you want. If the preference XI_PREF_ASSERT_ON_NULL_CID is on, the application will assert if the cid is invalid.

Return Value

Returns a pointer to the object with the given control ID. If there is no object with that ID, then **XI_NULL_OBJ** is returned. The returned object can be used for various XI functions, but should not be freed or directly modified.

Example

The following code is from “lstlink.c”.

```
static void update_numbers( XI_OBJ* list )
{
  XI_OBJ* column = xi_get_obj( list, COL_BASE_CID + LINK_NUM );

  if ( column != NULL )
    xi_cell_request( column );
}
```

xi_get_pref — Get a Preference

Summary

```
long xi_get_pref( XI_PREF_TYPE preftype );
```

Description

There are many preferences that can be set in XI using **xi_set_pref**. This function allows you to retrieve the current setting for those preferences.

preftype specifies the preference you want the value for. The various preferences are described in the section about **XI_PREF_TYPE**.

Return Value

Returns the value of the preference. You may wish to cast the long to a type that is appropriate for the preference. For example, **XI_PREF_3D_LOOK** can be cast to a **BOOLEAN**.

See Also

XI_PREF_TYPE, **xi_set_pref**

Example

The following code is from “lstdb.c”.

```
RCT r;

xi_get_def_rect( itfdef, (XI_RCT*) &r );
xvt_rect_offset( &r, (short)xi_get_pref( XI_PREF_ITF_MIN_LEFT ),
                 (short)xi_get_pref( XI_PREF_ITF_MIN_TOP ) );
itfdef->v.itf->win = xvt_win_create( W_DOC, &r, "Employee List",
                                     MENU_BAR_RID, TASK_WIN,
                                     WSF_SIZE | WSF_CLOSE
                                     | WSF_ICONIZABLE, EM_ALL,
                                     (EVENT_HANDLER)xi_event, 0L );
```

xi_get_rect — Get the Bounding Rectangle for an Object

Summary

```
XinRect* xi_get_rect( XI_OBJ *xi_obj, XinRect *rctp );
```

Description

This function returns the bounding rectangle of an object, in pixels. You can use **xi_get_def_rect** to get the bounding rectange for an object definition. If you want form units instead of pixels, you can call **xi_pu_to_fu** to convert the rectangle returned from this function. For interface objects, you can also call **xi_get_xi_rct** to get a rectangle in form units.

xi_obj is the object for which you want to get the bounding rectangle.

rctp points to a rectangle which will be filled in with the bounding rectangle of the object. This pointer is also returned from the function so that it can be used as an argument to some other function.

Return Value

Returns the value of **rctp**. The rectangle pointed to by **rctp** is filled in with the bounding rectangle of the object, in pixels.

Example

The following code is from “lstdb.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_XVT_POST_EVENT:
      if ( xiev->v.xvte.type == E_MOUSE_UP )
      {
        LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

        if ( list_info->list_resizing )
        {
          RCT    rct;
          WINDOW win = xi_get_window( itf );

          xi_get_rect( itf, (XI_RCT *)&rct );
          xvt_vobj_translate_points( win,
xvt_vobj_get_parent( win ),
                                     (PNT*)&rct, 2 );
          xvt_vobj_move( win, &rct );
        }
      }
      break;
  }
}
```

xi_get_sel — Get the Current Text Selection in an Object

Summary

```
void xi_get_sel( XI_OBJ *xi_obj, int *selstart, int *selstop );
```

Description

This function retrieves the current selected part of the text for an object.

xi_obj is the object for which you want to get the current text selection. This only works for edit fields and cells.

selstart points to an integer which will be filled with the starting position of the text selection. For example, if the first character is selected, the starting position will be zero.

selstop points to an integer which will be filled with the ending position of the text selection. For example, if only the first character is selected, the ending position will be one.

If the starting and ending positions are the same, then there is actually no current selection and these values reflect the position of the insertion point. For example, if the insertion point is at the beginning of the text, both starting and ending positions will be zero.

xi_get_text Get the Text for an Object

Summary

```
char* xi_get_text( XI_OBJ *xi_obj, char *s, int len );
```

Description

This function gets the text of an object. The text for an object is usually initialized in its definition, but may be edited by the user or set with **xi_set_text**.

xi_obj is the object for which you want to get text. For buttons, the text appears in the button. For cells, the text appears in the cell and can be edited. For columns, the text appears as the heading for the column. For edit fields, the text appears in the edit field and can be edited. For static text, the text is what is displayed. For interfaces, the text is the title for the window.

s points to a character array, which will be filled in with the text of the specified object. This value can be **NULL** if you do not want this function to copy the text to your own buffer. In that case, the returned value will refer to the actual buffer internal to XI. The string in that buffer **MUST NOT** be changed. However, this allows you to allocate your own buffer that can hold the text.

len contains the size of **s**. If the actual text for the object is longer, it will be truncated to this length.

Return Value

If **s** is not **NULL**, the function returns **s**. Otherwise, a pointer to the actual internal buffer for the text is returned. You should not free or change the text string in that case and you should copy the string if you want to save the value. Do not save just the pointer.

Example

The following code is from “lstdb.c”.

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_OFF_FIELD:
      if ( form_info->field_changed )
      {
        XI_OBJ* field = xiev->v.xi_obj;

        if ( !emp_set_value( form_info->handle,
                             field->cid - FIELD_BASE_CID,
                             xi_get_text( field, NULL, 0 ) ) )
        {
          xiev->refused = TRUE;
          xi_set_sel( field, 0, INT_MAX );
        } else
        {
          form_info->changed = TRUE;
          form_info->field_changed = FALSE;
          set_field_text( field, form_info->handle );
        }
      }
      break;
  }
}
```

xi_get_visible_columns — Get the Visible Columns for a List

Summary

```
void xi_get_visible_columns( XI_OBJ *list, int *first_vis,
                             int *last_vis );
```

Description

In a horizontal scrolling list, all columns that are in the horizontal scrolling portion of the list may not be visible at once. This functions allows you to get information about which columns are currently visible.

list is the list object that you want to get this information for. This function works only with list objects.

first_vis points to an integer which is filled with the index of the first visible column. If this pointer is **NULL**, this value will not be set.

last_vis points to an integer which is filled with the index of the last completely visible column. If this pointer is **NULL**, this value will not be set.

The values set by this function do not include fixed columns.

xi_get_visible_rows — Get the Visible Rows for a List

Summary

```
int xi_get_visible_rows( XI_OBJ *list, int *first_vis,
                         int *last_vis );
```

Description

This function retrieves information about which rows are currently visible in a list. If you are not using **get_all_records** on a list, these values will closely match the list of current records handles which is available from the **xi_get_list_info** function. In that case, only one or two of those handles may not be visible. If you are using **get_all_records**, this function will tell you which record handles correspond to visible rows.

list is the list object that you want to get this information for. This function works only with list objects.

first_vis points to an integer which is filled with the index of the first entirely visible row. This is a relative row number that can be translated to a record handle by indexing the array returned from **xi_get_list_info**. If this pointer is **NULL**, this value will not be set.

last_vis points to an integer which is filled with the index of the last entirely visible row. This is also a relative row number that can be translated to a record handle by indexing the array returned from **xi_get_list_info**. If this pointer is **NULL**, this value will not be set.

Return Value

Returns a count of the number of visible rows.

xi_get_window — Get the XVT Window for an Interface

Summary

```
WINDOW xi_get_window( XI_OBJ *itf );
```

Description

This function returns the XVT **WINDOW** handle that contains the specified interface.

itf is the XI interface object. Any other kind of object will cause unpredictable results.

Return Value

Returns an XVT window handle.

Example

The following code is from “lstdb.c”.

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  switch ( xiev->type )
  {
    ...
    case XIE_XVT_POST_EVENT:
      if ( xiev->v.xvte.type == E_MOUSE_UP )
      {
        LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

        if ( list_info->list_resizing )
        {
          RCT    rct;
          WINDOW win = xi_get_window( itf );

          xi_get_rect( itf, (XI_RCT*)&rct );
          xvt_vobj_translate_points( win,
xvt_vobj_get_parent( win ),
                                     (PNT*)&rct, 2 );
          xvt_vobj_move( win, &rct );
        }
      }
      break;
  }
}
```

xi_get_xi_rct — Get the Size of the Interface in Form Units

Summary

```
XinRect* xi_get_xi_rct( XI_OBJ *itf, XinRect *xi_rct );
```

Description

This function gets the size of a window that contains the specified interface, in form units.

itf is the interface object.

xi_rct is a pointer to the rectangle that will be set to the size of the interface, in form units.

Return Value

Returns **xi_rct** so that the result of this call can easily be passed to another function.

xi_init — Initialize XI

Summary

```
void xi_init( void );
```

Description

This function initializes XI and should be called before the first call to **xi_create**.

Example

The following code is from “main.c”.

```
static void init_application( void )
{
  XVT_FNTID font;

  xi_set_pref( XI_PREF_3D_LOOK, (long)THREE_DIMENSIONAL );
  xi_set_pref( XI_PREF_NATIVE_CTRLS, (long)!THREE_DIMENSIONAL );
  font = xvt_font_create();
  xvt_font_set_family( font, XVT_FFN_HELVETICA );
#if (XIWS == MTFWS || XIWS == XOLWS)
  xvt_font_set_size( font, 12 );
#elif (XIWS == PMWS)
  xvt_font_set_size( font, 8 );
#else
  xvt_font_set_size( font, 9 );
#endif
  xi_set_font_id( font );
  xvt_font_destroy( font );
  xi_init();
}
```

xi_insert_row — Insert a Row in a List

Summary

```
BOOLEAN xi_insert_row( XI_OBJ *list, int row );
```

Description

This function causes a row to be inserted by scrolling all the rows below downward. You should be sure that you can respond correctly to any **XIE_GET_*** events before calling this function. For example, if your internal structure is based on a database file, you should add the record to the database and then call **xi_insert_row**.

list is the list object into which the row will be inserted.

row is the relative row number that the new row will be inserted before. For example, a **row** of zero will insert a new row at the top of the list and a **row** of one will insert after the first row. A value of **INT_MAX** for **row** will insert a row at the bottom of the list.

Return Value

Returns **TRUE** if the row was inserted. **xi_insert_row** attempts to move the focus to the interface. If any focus change events were refused, then **xi_insert_row** would return **FALSE**.

Example

The following code is from “lstlink.c”.

```
static void process_button( XI_OBJ* itf, XI_OBJ* button )
{
  LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

  if ( button->type == XIT_CELL )
  {
    create_who_list( button, list_eh );
    return;
  }
```

```
  switch ( button->cid )
  {
    ...
    case ADD_ONE_CID:
    {
      XI_OBJ* obj = xi_get_focus( itf );

      if ( obj->type == XIT_CELL )
      {
        add_records( list_info->link_list, row_to_record(
                     obj->parent, obj->v.cell.row ), 1 );
        xi_insert_row( obj->parent, obj->v.cell.row + 1 );
      } else
      {
        add_records( list_info->link_list, 0L, 1 );
        refresh_list( xi_get_obj( itf, LIST_CID ) );
      }
      update_numbers( xi_get_obj( itf, LIST_CID ) );
      break;
    }
    ...
  }
}
```

xi_is_auto_tab

Summary

```
BOOLEAN xi_is_auto_tab( XI_OBJ *xi_obj );
```

Description

Returns **TRUE** if the specified object has the “auto-tab” option set.

xi_obj is an edit field or column object.

Return Value

Returns **TRUE** if **xi_obj** has “auto-tab” specified for it.

xi_is_checked Get Checked State of a Button

Summary

```
BOOLEAN xi_is_checked( XI_OBJ *xi_obj );
```

Description

This function returns the checked state of a radio button, check box or tab button object.

xi_obj is the button object.

Return Value

Returns the state of the specified radio button, check box, or tab button.

xi_is_focus_moving

Summary

```
BOOLEAN xi_is_focus_moving( XI_OBJ *itf );
```

Description

Returns **TRUE** if the focus is in the process of being moved.

itf is the interface object.

Return Value

Returns **TRUE** if the focus is in the process of being moved.

xi_is_itf — Determine if a Pointer is a Valid Interface

Summary

```
BOOLEAN xi_is_itf( XI_OBJ *itf );
```

Description

This function determines if a pointer is a valid XI interface without dereferencing the pointer. The function looks through the current list of XI interfaces, comparing against the interfaces in the list.

This function is useful if you need to perform some action on an XI object in an interface, and the interface may possibly have been deleted.

itf is the interface object that you want to check for validity.

Return Value

Returns **TRUE** if **itf** is a valid XI interface.

xi_is_window — Does the XVT WINDOW Contain an Interface

Summary

```
BOOLEAN xi_is_window( WINDOW win );
```

Description

This function determines if the window contains an XI interface.

win is an XVT window handle.

Return Value

Returns **TRUE** if **win** contains an XI interface.

xi_list_def_get_client_height

Summary

```
int xi_list_def_get_client_height( XI_OBJ_DEF* list_def, int rows );
```

Description

This function calculates the height, in pixels, of a list based on the specified definition and the desired number of rows. This height does not include a horizontal scroll bar.

list_def is the list definition that you want the height calculated for.

rows is the number of rows that will fit into that height.

Return Value

Returns the height in pixels.

xi_list_def_get_client_width

Summary

```
int xi_list_def_get_client_width( XI_OBJ_DEF* list_def,
                                  int columns );
```

Description

This function calculates the width, in pixels, of a list based on the specified definition and the desired number of columns. These columns must be defined for the list. This width does not include a vertical scroll bar.

list_def is the list definition that you want the width calculated for.

columns is the number of columns that will fit into that width.

Return Value

Returns the width in pixels.

xi_list_def_get_outer_height

Summary

```
int xi_list_def_get_outer_height( XI_OBJ_DEF* list_def, int rows );
```

Description

This function calculates the height, in pixels, of a list based on the specified definition and the desired number of rows. This height includes the horizontal scroll bar, if any. The result of this function can be used for the pixel rectangle in the list definition.

list_def is the list definition that you want the height calculated for.

rows is the number of rows that will fit into that height.

Return Value

Returns the height in pixels.

xi_list_def_get_outer_width

Summary

```
int xi_list_def_get_outer_width( XI_OBJ_DEF* list_def,
                                 int columns );
```

Description

This function calculates the width, in pixels, of a list based on the specified definition and the desired number of columns. These columns must be defined for the list. This width does includes the vertical scroll bar, if any.

list_def is the list definition that you want the width calculated for.

columns is the number of columns that will fit into that width.

Return Value

Returns the width in pixels.

xi_list_def_get_rows

Summary

```
int xi_list_def_get_rows( XI_OBJ_DEF* list_def );
```

Description

This function calculates the number of rows that will fit in the specified list definition. It does not include the partially displayed row, if any.

list_def is the list definition for which the calculation will be done.

Return Value

Returns the number of rows that will fit in the list.

XI_MAKE_CELL Macro to Make a Cell Object

Summary

```
#define XI_MAKE_CELL(objp, listobj, row_nbr, column_nbr) \
  memset((char *)objp, '\0', (size_t)sizeof(XI_OBJ)), \
  ((objp)->itf = (listobj)->parent, \
  (objp)->parent = listobj, \
  (objp)->type = XIT_CELL, \
  (objp)->v.cell.row = row_nbr, \
  (objp)->v.cell.column = column_nbr, \
  (objp)->nbr_children = 0)
```

Description

XI_MAKE_CELL is a useful macro for initializing an **XI_OBJ** structure with cell information. Cell objects are not kept in a list internal to XI. They also do not have individual control ID's, which you could pass to **xi_get_obj** to get an object pointer. However, you can use this macro to create a cell object that you can use to get or set text, get or set the current selection, or to call any other function that requires a pointer to an object.

To use this macro, first declare a structure of type **XI_OBJ**. Often, this will be a local variable in your function, as this object's scope does not need to extend beyond a call to an XI function. After you have declared a structure of type **XI_OBJ**, use the macro **XI_MAKE_CELL**, passing in the indicated arguments. **XI_MAKE_CELL** clears the memory of the object, and sets all fields in the structure that must be set.

objp points to the structure of type **XI_OBJ** that you declare in your function.

listobj points to the list object that contains the cell that you want to reference. You could get **listobj** by calling **xi_get_obj** and passing in the control ID of the list.

row_nbr is the row number of the cell. This is a relative row number based on visible rows. If you have a record handle and need to translate that to a row number, you should call **xi_get_list_info** to get the record handle array and then scan through for a match to your handle. The index number of that array is the appropriate row number. If the handle is not in that array, then the row is not currently visible and XI knows nothing about it.

column_nbr is the column number of the cell. Since columns may be moved, you may need to get the list of columns for a list by calling **xi_get_member_list** for the list and then scanning through for the matching column ID in the list of columns. The index number in that array is the appropriate column number.

Example

The following code is from "lstmem.c".

```
static void list_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_SELECT:
      switch ( xiev->v.select.xi_obj->type )
      {
        case XIT_ROW:
        {
          XI_OBJ* list = xiev->v.select.xi_obj->parent;
          int row = xiev->v.select.xi_obj->v.row_data.row;

          if ( column_to_code( list, xiev->v.select.column )
               == VALUE_IN_STOCK )
          {
            XI_OBJ cell;

            mem_change_stock( row_to_record( list, row ) );
            XI_MAKE_CELL( &cell, list, row, xiev->v.select.column );
            xiev->refused = TRUE;
            xi_cell_request( &cell );
          } else
            mem_select_row( row_to_record( list, row ),
                            xiev->v.select.selected );
          break;
        }
        ...
      }
    ...
  }
}
```

XI_MAKE_ROW Macro to Make a Row Object

Summary

```
#define XI_MAKE_ROW(objp, listobj, row_nbr) \
  memset((char *)objp, '\0', (size_t)sizeof(XI_OBJ)), \
  ((objp)->itf = (listobj)->parent, \
  (objp)->parent = listobj, \
  (objp)->type = XIT_ROW, \
  (objp)->v.row = row_nbr, \
  (objp)->nbr_children = 0)
```

Description

XI_MAKE_ROW is a useful macro for initializing an **XI_OBJ** structure with row information. Row objects are not kept in a list internal to XI. They also do not have individual control ID's, which you could pass to **xi_get_obj** to get an object pointer. However, you can use this macro to create a row object that you can use to delete a row.

To use this macro, first declare a structure of type **XI_OBJ**. Often, this will be a local variable in your function, as this object's scope does not need to extend beyond a call to an XI function. After you have declared a structure of type **XI_OBJ**, use the macro **XI_MAKE_ROW**, passing in the indicated arguments. **XI_MAKE_ROW** clears the memory of the object, and sets all fields in the structure that must be set.

objp points to the structure of type **XI_OBJ** that you declare in your function.

listobj points to the list object that contains the row that you want to reference. You could get **listobj** by calling **xi_get_obj** and passing in the control ID of the list.

row_nbr is the row number of the row. This is a relative row number based on visible rows. If you have a record handle and need to translate that to a row number, you should call **xi_get_list_info** to get the record handle array and then scan through for a match to your handle. The index number of that array is the appropriate row number. If the handle is not in that array, then the row is not currently visible and XI knows nothing about it.

Example

The following code is from "lstlink.c".

```
static long get_unselected_handle( XI_OBJ* list )
{
  int      count, num;
  long*    handles;

  handles = xi_get_list_info( list, &count );
  for ( num = 0; num < count; num++ )
  {
    XI_OBJ row;

    XI_MAKE_ROW( &row, list, num );
    if ( !( xi_get_attrib( &row ) & XI_ATR_SELECTED ) )
      return handles[ num ];
  }
  return 0;
}
```

xi_move_column Move a Column

Summary

```
void xi_move_column( XI_OBJ *column, int position );
```

Description

This function changes a column's position in a list. If the position is on the border between the fixed and horizontal scrolling portion of a list, the column will stay in the section that it was in. If that is not the desired result, you should call **xi_set_fixed_columns** after moving the column.

column is the column object to be moved.

position is the new position for the column. For example, zero will move it to be the first column in the list.

xi_move_focus — Move the Focus From One Object to Another

Summary

```
BOOLEAN xi_move_focus( XI_OBJ *xi_obj );
```

Description

This function moves the focus from one XI object to another. All appropriate focus change events will be generated and, if any are refused, the focus will not move. Moving the focus to an interface object causes all appropriate off focus events, but does not generate any on focus events.

It is common to move the focus to the interface when the user presses a "Save" or "OK" button. This causes the focus events to be generated so that all data can be validated before any action is taken for the button.

You can use **xi_set_focus** to change focus without generating focus events.

xi_obj is the object to which you want to move focus. The most common objects to move focus to are edit fields and cells. Buttons are valid on all platforms except the Macintosh. Moving focus to button containers, lists or forms will move focus to the first control within that object that can actually get focus.

Return Value

Returns **TRUE** if the focus moved successfully. Returns **FALSE** if any of the focus change events were refused.

See Also

xi_set_focus

Example

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_BUTTON:
      switch ( xiev->v.xi_obj->cid )
      {
        case ADD_BTN_CID:
          if ( !xi_move_focus( itf ) )
            break;
          if ( form_info->changed )
          {
            if ( emp_insert( form_info->handle ) )
              xvt_note( "Record added." );
            else
              xvt_note( "Add failed - error in file." );
            refresh_list( xi_get_obj( form_info->list_itf,
                          LIST_CID ) );
            form_info->changed = FALSE;
          }
          break;
      }
  }
}
```

xi_pu_to_fu Convert Pixel Units to Form Units

Summary

```
void xi_pu_to_fu( XI_OBJ *itf, PNT *pnts, int nbr_pnts );
```

Description

This function converts the coordinates in an array of points from pixels to form units. Note that because a rectangle can be cast into an array of points with two elements, it is possible to call this function to convert coordinates in a rectangle from pixels to form units.

itf is used to get the form unit metrics, based on the font for the interface, for the conversion.

pnts points to an array of points to be converted. If you have only one point to convert, you should pass the address of that point.

nbr_pnts is the number of points to be converted in the **pnts** array.

See Also

xi_fu_to_pu

xi_scroll Scroll a List

Summary

```
int xi_scroll( XI_OBJ *xi_obj, int nbr_rows );
```

Description

Uisng this function, a list can be scrolled a specified number of rows up or down, or it can be scrolled to the top or bottom, or it can be scrolled up or down one page.

The focus is removed from the list during the scrolling operation. Thus, when **xi_scroll** is called, there will be a side effect of XI generating appropriate **XIE_OFF_*** events to remove the focus from the list. This is done because an application may not know how many rows it can scroll before reaching the end of a database file. Removing the focus from the list before scrolling avoids the problem of not knowing whether a cell with the focus will be scrolled off the top or bottom of the list. Instead, the focus is always removed during scrolling and put back after scrolling.

You should not specify a scroll of more rows than XI knows about (from **xi_get_list_info**). If you need to scroll to an entirely different place in your data, use **xi_scroll_rec** or **xi_scroll_percent**.

xi_obj is the list object.

nbr_rows can be a positive number which causes that many lines to be scrolled onto the list from the bottom (this generates **XIE_GET_NEXT** events). If this value is negative, that many rows (converted to a positive number) will be scrolled onto the list from the top. There are four special values that can be passed in this argument:

> **XI_SCROLL_PGUP**: Will cause the list to scroll up one page so that new rows appear from the top of the list. The number of overlapping rows is determined by **XI_PREF_OVERLAP**. This will cause **XIE_GET_NEXT** events for the list.
>
> **XI_SCROLL_PGDOWN**: Will cause the list to scroll down one page so that new rows appear from the bottom of the list. The number of overlapping rows is determined by **XI_PREF_OVERLAP**. This will cause **XIE_GET_PREV** events for the list.
>
> **XI_SCROLL_FIRST**: Will cause the list to scroll to the first rows in the list. This will cause **XIE_GET_FIRST** and **XIE_GET_NEXT** events for the list.
>
> **XI_SCROLL_LAST**: Will cause the list to scroll to the last rows in the list. This will cause **XIE_GET_LAST** and **XIE_GET_PREV** events for the list.

Return Value

For **XI_SCROLL_FIRST** and **XI_SCROLL_LAST**, the return value should be ignored. In all other cases, the actual number of rows that were scrolled onto the list is returned. This may be less than the requested number of rows if the list reaches the top or bottom of all the rows.

See Also

XI_SCROLL_*, **xi_scroll_rec**, **xi_scroll_percent**

Example

The following code is from "lstdb.c".

```
static void refresh_list( XI_OBJ* list )
{
  int      count;
  long*    handles;

  handles = xi_get_list_info( list, &count );
  if ( count == 0 )
    xi_scroll( list, XI_SCROLL_FIRST );
  else
    emp_scroll_rec( list, handles[0] );
}
```

xi_scroll_percent Scroll a List to a Percentage

Summary

```
int xi_scroll_percent( XI_OBJ *list, int percent );
```

Description

This function scrolls the list by generating an **XIE_GET_FIRST** event with the specified percentage, followed by **XIE_GET_NEXT** events for the rest of the rows. If your list has more than 100 total records in it, you will not be able to scroll to any particular record by using just a percentage. For this reason, we suggest using **xi_scroll_rec** instead, even in smaller lists.

list is the list object to be scrolled.

percent is the percentage passed to the **XIE_GET_FIRST** event.

Return Value

Returns the relative row number for the record at the specified percentage. This will usually be zero, but if there are not enough records after the requested record to fill list, then previous records will be requested. For example, if you call this function with a percentage of 80 and two records had to be requested before the record returned from the **XIE_GET_FIRST** event, the function returns two (the two previous records are 0 and 1).

See Also

xi_scroll_rec, **xi_scroll**

xi_scroll_rec Scroll a List to a Given Record

Summary

```
int xi_scroll_rec( XI_OBJ *list, long record, XinColor row_color,
                   unsigned long attrib, int row_height );
```

Description

This function scrolls the list to a specific record. This call generates **XIE_GET_NEXT** events for the list. It may also generate **XIE_GET_PREV** events if the specified record is near the bottom of the list and there are not enough records below to fill the list.

This function can be called during the **XIE_INIT** event for the interface to position the list. This will prevent an **XIE_GET_FIRST** event from being generated as it usually would during the initialization of a list.

The focus is removed from the list during the scrolling operation. Thus, when **xi_scroll_rec** is called, there may be a side effect of XI generating appropriate **XIE_OFF_*** events to remove the focus from the list.

list is the list object to be scrolled.

record is the record handle. This takes the place of the **data_rec** member of the **XIE_GET_*** events. It will be used for cell request events for that row.

row_color is the color for all cells in the row. Use zero for the default colors.

attrib is an OR'ed combination of **XI_ATR_*** values which control some of the behavior of the row.The following attributes apply to rows:

```
XI_ATR_SELECTED
```

row_height is the desired height, in pixels, for the row. Use zero for the default row height.

Return Value

Returns the relative row number for the specified record. This will usually be zero, but if there are not enough records after the requested record to fill list, then previous records will be requested. For example, if two records had to be requested before the specified record, the function returns two (the two previous records are 0 and 1).

See Also

xi_scroll_percent, **xi_scroll**

Example

The following code is from "lstmem.c".

```
static void select_set_font( XI_OBJ* itf, XVT_FNTID font )
{
  XI_OBJ* list = xi_get_obj( itf, LIST_CID );
  LIST_INFO* list_info = (LIST_INFO*)xi_get_app_data(itf);

  ...

  {
    int count;
    long* handles;
    XI_EVENT xiev;

    handles = xi_get_list_info( list, &count );
    memset( (char*)&xiev, 0, sizeof( xiev ) );
    mem_rec_request( list_info->records + (int)handles[0], &xiev );
    xi_scroll_rec( list, handles[0], (COLOR)0,
                   xiev.v.rec_request.attrib,
                   xiev.v.rec_request.row_height );
  }
}
```

xi_set_app_data — Set Application Data Associated with an XI Object

Summary

```
void xi_set_app_data( XI_OBJ *xi_obj, long app_data );
```

Description

This function sets the application data for an XI object. You can set the application data for an interface in the interface definition. The application data can be retrieved later using the **xi_get_app_data** function.

A common use of application data is to put a pointer to an event handler in the application data or to put a pointer to an event handler in a structure pointed to by the application data. Then, when events come in to an XI event handler, call this event handler indirectly. This allows you to have an event handler for every control, or for types of controls.

Another common use for application data is to store the address of a C++ object that corresponds to the control. Events can then be dispatched to member functions of that object using that pointer.

xi_obj points to the object for which you want to set the application data. Rows and cells are not valid.

app_data is the application data. If your application data is a pointer to a structure, you should use the XVT macro **PTR_LONG** to convert it to a long.

See Also

xi_get_app_data

Example

The following code is from “lstdb.c”.

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
  }
}
```

xi_set_app_data2 Internal Function

Summary

```
void xi_set_app_data2( XI_OBJ *xi_obj, long app_data2 );
```

Description

This function is for internal use only.

xi_set_attrib Set Object Attributes

Summary

```
void xi_set_attrib( XI_OBJ *xi_obj, unsigned long attrib );
```

Description

This function allows you to modify attributes for an XI object after the object has been created. You can modify the state of most of the attributes that you can set when you create an object.

You cannot set attributes for an object that has the focus.

Often, you will want to set or clear one attribute for an object. To set one attribute, you will need to bitwise OR it with the current attributes, as returned by **xi_get_attrib**. To clear one attribute, you will need to bitwise AND its complement (~ operator) with the current attributes.

xi_obj is the object for which you want to set attributes.

attrib contains the new attributes of the object.

See Also

XI_ATR_*

Example

The following code, from “lstlink.c”, demonstrates making controls visible or invisible.

```
static void change_section( SECTION_INFO* section, BOOLEAN flag )
{
  int      num;
  XI_OBJ** obj;

  for ( num = 0, obj = section->objs; num < section->count;
        num++, obj++ )
    if ( flag )
      xi_set_attrib( *obj, xi_get_attrib( *obj ) | XI_ATR_VISIBLE );
    else
      xi_set_attrib( *obj, xi_get_attrib( *obj ) &
~XI_ATR_VISIBLE );
}
```

xi_set_bufsize Set the Buffer Size of an Object

Summary

```
void xi_set_bufsize( XI_OBJ *xi_obj, int size );
```

Description

This function changes the buffer size used by the object for editing. The buffer size is the number of characters that can be typed into an edit field or cell. The buffer size for cells can only be set for all cells in a column.

It is invalid to call **xi_set_bufsize** on an object that has the focus. Move the focus to the interface if necessary.

xi_obj is the object whose buffer size will be changed. Edit fields and columns are the only objects that have buffer size. However, passing a form as the object will set the buffer size for all of its edit fields and passing a list as the object will set the buffer size for all of its columns.

size is the number of characters that can be displayed and edited.

xi_set_column_width Set the Width of a Column

Summary

```
void xi_set_column_width( XI_OBJ *column, int width );
```

Description

This function changes the width of a column in a list. No cell in the column should have focus when this function is called. If a cell has the focus, you can move the focus to the interface, call this function, and then move it back to the cell.

column is the column whose width will be changed.

width is the new width of the column in form units.

xi_set_color Set Colors for an Object

Summary

```
void xi_set_color( XI_OBJ *xi_obj, XI_COLOR_PART part,
                   XinColor color );
```

Description

This function sets the color for the specified part of an object. This function should not be called on an object that has the focus. If necessary, move the focus to the interface before calling this function.

xi_obj contains a pointer to the object for which you want to change the color.

part is a member of the **XI_COLOR_PART** enum, and specifies the part of the object for the new color. The valid parts varies for each object.

For edit fields, **XIC_ENABLED**, **XIC_BACK**, **XIC_HILIGHT**, **XIC_ACTIVE**, **XIC_ACTIVE_BACK**, **XIC_SHADOW**, **XIC_DISABLED** and **XIC_DISABLED_BACK** are valid.

For cells, lines and static text, **XIC_ENABLED** and **XIC_BACK** are valid.

For rows and buttons only **XIC_ENABLED** is valid.

For lists, **XIC_ENABLED**, **XIC_BACK**, **XIC_ACTIVE**, **XIC_ACTIVE_BACK**, **XIC_DISABLED**, **XIC_DISABLED_BACK** and **XIC_WHITE_SPACE** are valid.

For rectangles, **XIC_ENABLED**, **XIC_BACK**, **XIC_HILIGHT**, and **XIC_SHADOW** are valid.

For the interface, only **XIC_BACK** is valid.

color is the new color.

xi_set_fixed_columns

Summary

```
void xi_set_fixed_columns( XI_OBJ *list, int fixed_columns );
```

Description

This function changes the number of fixed columns for an existing list.

list is the list object that you wish to change.

fixed_columns is the new number of fixed columns for **list**.

xi_set_focus Set the Focus to an Object

Summary

```
void xi_set_focus( XI_OBJ *xi_obj );
```

Description

This function sets the focus to an object. No focus events will be generated and the focus will always be moved to that object. In general, you will want to use **xi_move_focus** so that focus events will be generated.

xi_obj is the object to which you want the focus to be set. The most common objects to set focus to are edit fields and cells. Buttons are valid on all platforms except the Macintosh. Moving focus to button containers, lists or forms will move focus to the first control within that object that can actually get focus.

See Also

xi_move_focus

xi_set_font_id

Summary

```
void xi_set_font_id( XVT_FNTID font_id );
```

Description

This function sets the default font for all interfaces. It must be called before **xi_init** is called.

font_id is the XVT font that will be used as the default for all interfaces. The font for an interface can be overridden by setting the **font_id** member of the interface definition. You must create the font using XVT functions. XI copies the font, so you can destroy it as soon as this function returns.

Example

The following code is from “main.c”.

```
static void init_application( void )
{
  XVT_FNTID font;

  xi_set_pref( XI_PREF_3D_LOOK, (long)THREE_DIMENSIONAL );
  xi_set_pref( XI_PREF_NATIVE_CTRLS, (long)!THREE_DIMENSIONAL );
  font = xvt_font_create();
  xvt_font_set_family( font, XVT_FFN_HELVETICA );
#if (XIWS == MTFWS || XIWS == XOLWS)
  xvt_font_set_size( font, 12 );
#elif (XIWS == PMWS)
  xvt_font_set_size( font, 8 );
#else
  xvt_font_set_size( font, 9 );
#endif
  xi_set_font_id( font );
  xvt_font_destroy( font );
  xi_init();
}
```

xi_set_fore_color

Summary

```
void xi_set_fore_color( XI_OBJ *xi_obj, XinColor color );
```

Description

This function is made obsolete by the function **xi_set_color**. It is equivalent to calling that function for the **XIC_ENABLED** part.

xi_set_list_size Change the Size of a List

Summary

```
void xi_set_list_size( XI_OBJ *list, int height, int width );
```

Description

This function dynamically changes the size of a list.

list is the list object to be changed.

height is the new height, in pixels.

width is the new width, in pixels.

Example

The following code is from “lstsync.”.

```
case XIE_XVT_EVENT:
  if ( xiev->v.xvte.type == E_SIZE )
  {
    XI_OBJ* list;
    RCT     rct;

    list = xi_get_obj( itf, LIST2_CID );
    xi_get_rect( list, (XI_RCT *)&rct );
    if ( rct.right != xiev->v.xvte.v.size.width )
      xi_set_list_size( list, rct.bottom - rct.top,
                        xiev->v.xvte.v.size.width - rct.left );

    list = xi_get_obj( itf, LIST3_CID );
    xi_get_rect( list, (XI_RCT *)&rct );
    if ( rct.bottom != xiev->v.xvte.v.size.height )
      xi_set_list_size( list, xiev->v.xvte.v.size.height - rct.top,
                        rct.right - rct.left );

    list = xi_get_obj( itf, LIST4_CID );
    xi_get_rect( list, (XI_RCT *)&rct );
    if ( rct.bottom != xiev->v.xvte.v.size.height
         || rct.right != xiev->v.xvte.v.size.width )
      xi_set_list_size( list, xiev->v.xvte.v.size.height - rct.top,
                        xiev->v.xvte.v.size.width - rct.left );
  }
  break;
```

xi_set_icon Set the Icon for an Object

Summary

```
void xi_set_icon( XI_OBJ *xi_obj, int icon_rid,
                  int down_icon_rid );
```

Description

This function sets the icon for an object.

xi_obj is the object that will have its icon or icons set. Only cells and buttons are valid. Other objects will be supported in a future release.

icon_rid is the resource ID for the icon. For cells, this isthe icon that appears in that cell. For buttons, it is the icon that appears when the button is not pressed.

down_icon_rid is the resource ID for the down icon. For buttons, it is the icon that appears while the button is pressed. It is ignored for cells.

xi_set_obj_font_id

Summary

```
void xi_set_obj_font_id( XI_OBJ *xi_obj, XVT_FNTID font_id );
```

Description

This function sets the font for an object.

xi_obj is the object to set the font for. Only cells, columns and interfaces are valid. Setting the font for a column will change the font for the heading text only.

font_id is the XVT font that will be used as the font for the specified object. You must create this font using XVT functions. Since XI copies the font, you can destroy it after this function returns.

xi_set_pref Set a Preference

Summary

```
void xi_set_pref( XI_PREF_TYPE preftype, long value );
```

Description

There are many preferences that can be set using this function.

preftype specifies the preference you want to set the value for. The various preferences are described in the section about **XI_PREF_TYPE**.

value is the new value for that preference.

See Also

XI_PREF_TYPE

Example

The following code is from “main.c”.

xi_set_rect Move a Object

Summary

```
void xi_set_rect( XI_OBJ *xi_obj, XinRect *xi_rct, BOOLEAN
do_invalidates );
```

Description

This function sets the row height, in pixels.

xi_obj contains a pointer to the object for which you want to change the position. Edit fields, containter, buttons, lists, text and rectangles are valid objects.

xi_rct is a pointer to the rectangle that contains the new position of the object, in form units.

do_invalidates if this BOOLEAN is TRUE, the window will be invalidated after the object is moved.

xi_set_row_height Set the Height of a Row

Summary

```
void xi_set_row_height( XI_OBJ *row, int height );
```

Description

This function sets the row height, in pixels.

row is the row to set the height for. Often, this object is created using the **XI_MAKE_ROW** macro.

height is the new height for the row, in pixels.

xi_set_sel Set the Current Text Selection in an Object

Summary

```
void xi_set_sel( XI_OBJ *xi_obj, int selstart, int selstop );
```

Description

This function sets the selected part of the text or the insertion point for an object. This function also causes focus to be moved to the specified object if it is not already there. If focus must be moved, all appropriate focus events will be generated. We suggest that you explicity call **xi_move_focus** or **xi_set_focus** to move focus to the object before calling this function. This will give you control over whether or not focus events are generated and also, if you use **xi_move_focus**, its return value indicates if any of the focus events were refused.

xi_obj is the object for which you want to set the selection. This only works for edit fields and cells.

selstart is the starting position of the text selection or, if **selstop** is the same value, then it is the position of the insertion point. For example, if **selstart** is one, the selection will begin with the second character in the text or, if **selstop** is also one, then the insertion point will be between the first and second characters.

selstop is the ending position of the text selection or, if **selstart** is the same value, then it is the position of the insertion point. For example, if **selstop** is two, text selection will end with the second character or, if **selstart** is also two, then the insertion point will be between the second and third characters.

Example

The following code is from “lstdb.c”.

```
static void form_eh( XI_OBJ* itf, XI_EVENT* xiev )
{
  FORM_INFO* form_info = (FORM_INFO*)xi_get_app_data( itf );

  switch ( xiev->type )
  {
    ...
    case XIE_OFF_FIELD:
      if ( form_info->field_changed )
      {
        XI_OBJ* field = xiev->v.xi_obj;

        if ( !emp_set_value( form_info->handle,
                             field->cid - FIELD_BASE_CID,
                             xi_get_text( field, NULL, 0 ) ) )
        {
          xiev->refused = TRUE;
          xi_set_sel( field, 0, INT_MAX );
        } else
        {
          form_info->changed = TRUE;
          form_info->field_changed = FALSE;
          set_field_text( field, form_info->handle );
        }
      }
      break;
  }
}
```

xi_set_text Set the Text for an Object

Summary

```
void xi_set_text( XI_OBJ *xi_obj, char *s );
```

Description

This function sets the text for an object.

xi_obj is the object for which you want to set the text. For buttons, the text appears in the button. For cells, the text appears in the cell and can be edited. For columns, the text appears as the heading for the column. For edit fields, the text appears in the edit field and can be edited. For static text, the text is what is displayed. For interfaces, the text is the title for the window.

s specifies the text to be set on the object.

Example

The following code is from "datlink.c".

```
void link_set_text( long handle, LINK_FIELD field, XI_OBJ* obj )
{
  xi_set_text( obj, get_text( handle, field ) );
}
```

xi_tree_dbg Output Tree Memory Summary

Summary

```
void xi_tree_dbg( char *title );
```

Description

This function will print a current summary of all tree memory usage. The output will include the listing of all memory still allocated at the time of the call. The "printing" is done using the XVT function **xvt_debug_printf**.

If you have XI source code, and the constant **TREEDEBUG** is defined at compile time, then all functions are redefined to use the **__LINE__** and **__FILE__** symbols. This information will be stored with each memory block, so that if corruption is detected, the file and line number will be available. The file and line information is also printed out by **xi_tree_dbg**. In order to use **xi_tree_dbg** with **TREEDEBUG** defined, you must recompile every module that uses the tree memory functions, including XI source code.

title is printed to the debug file before all of the summary information.

There are other attributes of the tree memory module that can be changed by defining constants in TREE.C source module:

#define PAD n	Pad all allocations by **n** bytes and initialize to known pattern. The initialization is done even if FENCE is undefined, so that the application can call **xi_tree_check_sanity** at any time.
#define FENCE n	Check padding bytes for known pattern on every nth call to a tree memory function
#define GRANULARITY n	Round up all allocations to multiple of n

xi_tree_check_sanity — Check for Corruption of the Heap

Summary

```
void xi_tree_check_sanity( char *title );
```

Description

This function looks through the entire tree memory data structure and looks for anything wrong, including fences that were stepped on. You must link with the debugging version of the XI library in order to call this function. The debugging version of the tree memory allocator calls **xi_tree_check_sanity** automatically on every tenth call to **xi_tree_malloc** and **xi_tree_free**.

title is printed to the debug file. The “printing” is done using the XVT function **xvt_debug_printf**.

xi_tree_free — Free Tree Memory

Summary

```
void xi_tree_free( void *p );
```

Description

This function frees the allocated memory and any other allocations that were parented to this memory. This process is done recursively so that memory parented to memory that was parented to **p** is freed, and so on.

p contains the pointer to be freed. This pointer must have been allocated with **xi_tree_malloc** or **xi_tree_realloc**.

Example

The following code is from “lstdb.c”.

```
xi_create( NULL, itfdef );
xi_dequeue();
xi_tree_free( itfdef );
```

xi_tree_get_parent — Get the Parent of an Allocated Object

Summary

```
void *xi_tree_get_parent( void *p );
```

Description

p is a pointer that was allocated with **xi_tree_malloc** or **xi_tree_realloc**.

Return Value

Returns the current parent of the allocated tree memory.

xi_tree_malloc — Allocate Tree Memory

Summary

```
void *xi_tree_malloc( size_t size, void *parent );
```

Description

This function allocates memory and associates it with the specified parent. All of the allocated memory will be initialized to zero.

size contains the number of bytes to be allocated.

parent specifies another memory allocation which will be the parent for this allocation. When that memory, or its parent (or its parent's parent, etc.) is freed, this memory will automatically be freed also. This pointer must have been allocated using **xi_tree_malloc** or **xi_tree_realloc**. If **NULL**, then this memory will have no parent and must be freed by calling **xi_tree_free** or have its parent changed at a later time using **xi_tree_reparent**.

Return Value

Returns a pointer to the allocated memory. This function never returns **NULL**. Instead, if **xi_tree_malloc** cannot get more memory, the function registered by a call to **xi_tree_reg_error_fcn** is called. By default, that error function will terminate the program by calling **xvt_fatal**.

Example

The following code is from “lstcol.c”.

```
void add_column_def( COLUMN_LIST_INFO* list_info,
                     XI_OBJ_DEF* definition )
{
  COLUMN_INFO* column_info;

  if ( list_info->nbr_columns == 0 )
    list_info->columns = (COLUMN_INFO*)xi_tree_malloc(
                         sizeof( COLUMN_INFO ), NULL );
  else
    list_info->columns = (COLUMN_INFO*)xi_tree_realloc(
                         list_info->columns, sizeof( COLUMN_INFO )
                         * ( list_info->nbr_columns + 1 ) );
  column_info = list_info->columns + list_info->nbr_columns++;
  column_info->column_definition = definition;
  column_info->selected = FALSE;
}
```

xi_tree_realloc Reallocate Tree Memory

Summary

```
void *xi_tree_realloc( void *p, size_t size );
```

Description

This function changes the size of a memory block that was allocated with **xi_tree_malloc** or a previous call to **xi_tree_realloc**. The parent remains unchanged.

p is a pointer previously allocated with **xi_tree_malloc** or **xi_tree_realloc**.

size is the desired new size for **p**.

Return Value

Returns the reallocated piece of memory. This function never returns **NULL**. Instead, if **xi_tree_realloc** cannot get more memory, the function registered by a call to **xi_tree_reg_error_fcn** is called. By default, that function will terminate the program by calling **xvt_fatal**.

Example

The following code is from “lstcol.c”.

```
void add_column_def( COLUMN_LIST_INFO* list_info,
                     XI_OBJ_DEF* definition )
{
  COLUMN_INFO* column_info;

  if ( list_info->nbr_columns == 0 )
    list_info->columns = (COLUMN_INFO*)xi_tree_malloc(
                         sizeof( COLUMN_INFO ), NULL );
  else
    list_info->columns = (COLUMN_INFO*)xi_tree_realloc(
                         list_info->columns, sizeof( COLUMN_INFO )
                         * ( list_info->nbr_columns + 1 ) );
  column_info = list_info->columns + list_info->nbr_columns++;
  column_info->column_definition = definition;
  column_info->selected = FALSE;
}
```

xi_tree_reg_error_fcn — Register a Function for Tree Memory Errors

Summary

```
void xi_tree_reg_error_fcn( void (*fcn)(void) );
```

Description

This function registers a new tree memory error handling function. **xi_tree_malloc** and **xi_tree_realloc** never return **NULL**. Instead, if they cannot allocate memory, they call the function registered by a call to **xi_tree_reg_error_fcn**. This means that you don't need code that tests for **NULL** after calling **xi_tree_malloc** or **xi_tree_realloc**, but you still can catch memory allocation errors by registering your own function.

fcn is a pointer to the function that will be called if memory cannot be allocated.

xi_tree_reparent — Change the Parent of an Allocated Piece of Tree Memory

Summary

```
void xi_tree_reparent( void *p, void *parent );
```

Description

This function changes the parent of an allocated block of tree memory. Memory that is parented to **p** will remain parented to **p** no matter how the parent of **p** changes.

p is a pointer to memory that was allocated with **xi_tree_malloc** or **xi_tree_realloc**.

parent is a pointer to the new parent for **p**. If this value is **NULL**, then **p** will have no parent and should be freed at some point by calling **xi_tree_free**.

xi_vir_pan Pan a Virtual Interface

Summary

```
void xi_vir_pan(XI_OBJ *itf, int delta_x, int delta_y );
```

Description

This function pans a virtual interface in any direction.

itf is the interface object that will be panned.

delta_x is the distance to pan horiztonally, in form units. Positive values will pan things on from the right onto the visible portion of the interface. Negative values will pan things on from the left.

delta_y is the distance to pan vertically, in form units. Positive values will pan things on from the bottom of the interface. Negative values will pan things on from the left