XCONTROL Function Reference

XCONTROL Callback Functions

The **XCONTROL** callback functions are user-supplied functions which are identified to the Control Panel in the **CPXINFO** structure returned by the **cpx_init()** function which is also described in this section. When creating a Form CPX, the only callback function that is utilized is **cpx_call()**. The remaining functions are used only when creating Event CPX's. The **XCONTROL** callback functions are:

- **cpx_button()**
- **cpx_call()**
- **cpx_close()**
- **cpx_draw()**
- **cpx_hook()**
- **cpx_init()**
- **cpx_key()**
- **cpx_m1()**
- **cpx_m2()**
- **cpx_timer()**
- **cpx_wmove()**

cpx_button()

VOID (*cpx_button)(***mrets, nclicks, event*** **)**
MRETS *****mrets***;**
WORD ***nclicks*****;**
WORD *****event***;**

cpx_button() is called in an Event CPX when a **MU_BUTTON** event has occurred.

PARAMETERS *mrets* points to a structure containing the mouse event which triggered the function as follows:

```
typedef struct
{
       WORD x;        /* X position of mouse */
       WORD y;        /* Y position of mouse */
       WORD buttons;  /* Mask of buttons depressed */
       WORD kstate;   /* Keyboard shift state */
} MRETS;
```

nclicks specifies the number of clicks processed. If this event should terminate the CPX, the function should place a 1 in the **WORD** pointed to by *event*.

BINDING

```
cpxinfo.cpx_button = cpx_button;

return ( &cpxinfo );
```

COMMENTS This function will only be called if **Set_Evnt_Mask()** is called with **MU_BUTTON** specified as an event to wait for.

cpx_call()

BOOLEAN (*cpx_call)(***work*** **)**
GRECT *****work***;**

cpx_call() is called immediately after the **cpx_init()** function when the user activates the CPX.

PARAMETERS Upon entry, the **GRECT** structure pointed to by work contains the current rectangular extent of the control panel window work area.

BINDING

```
cpxinfo.cpx_call = cpx_call;

return ( &cpxinfo );
```

RETURN VALUE The **cpx_call()** function should return **TRUE** if it wants to continue processing events through the event handlers specified in the **CPXINFO** structure or **FALSE** to indicate the CPX is finished.

COMMENTS When exiting the **cpx_call()** function, the CPX must deallocate any allocated memory and close any **VDI** workstations opened.

cpx_close()

VOID (*cpx_close)(***flag*** **)**
BOOLEAN ***flag*****;**

cpx_close() is called in an Event CPX when a **WM_CLOSED** or **AC_CLOSE** message is received by the control panel.

PARAMETERS *flag* contains **TRUE** if a **WM_CLOSED** message was received or **FALSE** if **AC_CLOSE** was received.

BINDING

```
cpxinfo.cpx_close = cpx_close;

return ( &cpxinfo );
```

COMMENTS This function will only be called if **Set_Evnt_Mask()** is called with **MU_MESAG** specified as an event to wait for.

WM_CLOSED messages should be treated as equivalent to ‘OK’ whereas **AC_CLOSE** messages should be treated as ‘Cancel’.

cpx_draw()

VOID (*cpx_draw)(***clip*** **)**
GRECT *****clip***;**

cpx_draw() is called when a **WM_REDRAW** message is received by the control panel in an Event CPX.

PARAMETERS *clip* points to a **GRECT** structure specifiying the dirtied area.

BINDING

```
cpxinfo.cpx_draw = cpx_draw;

return ( &cpxinfo );
```

COMMENTS This routine should utilize **GetFirstRect()** and **GetNextRect()** to obtain the true rectangles of the area to redraw.

This function will only be called if **Set_Evnt_Mask()** is called with **MU_MESAG** specified as an event to wait for.

cpx_hook()

BOOLEAN (*cpx_hook)(***event, msg, mrets, key, nclicks*** **)**
WORD ***event*****;**
WORD ***msg*****;**
WORD ***mrets*****;**
WORD ***key, nclicks*****;**

cpx_hook() is called in an Event CPX immediately after the Control Panel's **evnt_multi()** function returns before the message is processed.

PARAMETERS All parameters share counterparts with the **evnt_multi()** function. For a detailed explanation of the return values please consult the documentation for that function. *event* contains the event mask of one or more events that occurred. *msg* points to an array of eight **WORD**s containing the message buffer. *mrets* and *nclicks* point to the mouse event (if any) as described in **cpx_button()**. *key* points to a **WORD** containing the keyboard scancode of the key pressed (if any).

BINDING

```
cpxinfo.cpx_hook = cpx_hook;

return ( &cpxinfo );
```

RETURN VALUE The function should return **TRUE** to override default event handling or **FALSE** to continue processing the message.

cpx_init()

CPXINFO (*cpx_init)(***xcpb*** **)**
XCPB ***xcpb*****;**

cpx_init() is called upon bootup and every subsequent time the CPX is opened by the user.

PARAMETERS *xcpb* points to an XControl Parameter Block structure as described in the beginning of this chapter.

BINDING The **cpx_init()** function is called by JSR'ing to the first location in the CPX's TEXT segment. 'C' programmers should assemble and link the following code as the first object file in the link to ensure that the correct function is properly called:

```
; Startup stub for CPX's without save area

         .xref    _cpx_init

         .text

cpxstart:
         jmp      _cpx_init

         .end
```

If the CPX has default data which is to be saved back into the CPX with the **CPX_Save()** function, the following stub should be used (substitute the '.dc.w 1' statement with the appropriate amount of space required to store your data):

```
; Startup stub for CPX's with save area

         .xref    _cpx_init
         .globl   _save_vars

         .text

cpxstart:
         jmp      _cpx_init

         .data

_save_vars:
         .dc.w        1

         .end
```

RETURN VALUE The **cpx_init()** function returns a pointer to its **CPXINFO** structure to allow the Control Panel to access its other routines. If it is a 'Set-Only' CPX, it should return **NULL**.

COMMENTS A CPX can distunguish when a CPX is booting by checking the *xcpb->booting* structure member.

It is recommended that the CPX to create a copy of *xcpb* each time **cpx_init()** is called for the other callback functions to utilize.

cpx_key()

VOID (*cpx_key)(***kstate, key, event*** **)**
WORD ***kstate*;**
WORD ***key*;**
WORD ****event*;**

cpx_key() is called in an Event CPX when a **MU_KEYBD** event has occurred.

PARAMETERS *kstate* specifies the state of the keyboard shift keys as in **evnt_keybd()**. key specifies the keyboard scan code of the key struck. The **WORD** pointed to by *event* should be filled in with a 1 if this event should terminate the CPX.

BINDING

```
cpxinfo.cpx_key = cpx_key;

return ( &cpxinfo );
```

COMMENTS This function will only be called if **Set_Evnt_Mask()** is called with **MU_KEYBD** specified as an event to wait for.

cpx_m1()

VOID (*cpx_m1)(***mrets*, *event*** **)**
MRETS ***mrets*;**
WORD ***event*;**

cpx_m1() is called when a **MU_M1** event has occurred in an Event CPX.

PARAMETERS *mrets* will contain a pointer to a **MRETS** structure as specified in **cpx_button()** which contains the mouse state as it satisfied the condition. The **WORD** pointed to by event should be filled in with 1 if this event should terminate the CPX.

BINDING

```
cpxinfo.cpx_m1 = cpx_m1;

return ( &cpxinfo );
```

COMMENTS This function will only be called if **Set_Evnt_Mask()** is called with **MU_M1** specified as an event to wait for.

SEE ALSO **cpx_m2()**

cpx_m2()

VOID (*cpx_m2)(***mrets*, *event*** **)**
MRETS ***mrets*;**
WORD ***event*;**

cpx_m2() is called when a **MU_M2** event has occurred in an Event CPX.

PARAMETERS See **cpx_m1()**.

BINDING

```
cpxinfo.cpx_m2 = cpx_m2;

return ( &cpxinfo );
```

COMMENTS This function will only be called if **Set_Evnt_Mask()** is called with **MU_M2** specified as an event to wait for.

SEE ALSO **cpx_m1()**

cpx_timer()

VOID (*cpx_timer)(*event*)
WORD **event*;

cpx_timer() is called when a **MU_TIMER** event has occurred in an Event CPX.

PARAMETERS The **WORD** pointed to by event should be filled in with 1 if this event should terminate the CPX.

BINDING

```
cpxinfo.cpx_timer = cpx_timer;

return ( &cpxinfo );
```

COMMENTS This function will only be called if **Set_Evnt_Mask()** is called with **MU_TIMER** specified as an event to wait for.

cpx_wmove()

VOID (*cpx_wmove)(*work*)
GRECT **work*;

cpx_wmove() is called when a **WM_MOVED** message is received by the Control Panel in an Event CPX.

PARAMETERS *work* is a pointer to a **GRECT** containing the new coordinates of the window work area.

BINDING

```
cpxinfo.cpx_wmove = cpx_wmove;

return ( &cpxinfo );
```

COMMENTS This function will only be called if **Set_Evnt_Mask()** is called with **MU_MESAG** specified as an event to wait for.

XCONTROL Utility Functions

The **XCONTROL** utility functions are accessed via the **XCPB** (XControl Parameter Block) in the following format for users of ‘C’:

```
ret = (*xcpb->Function)( param1, param2, ... )
```

These functions provide functions useful mostly to CPX’s as well as functions that closely resemble **AES** functions better suited for CPX’s. The **XCONTROL** Utility Functions are:

- **(*xcpb->CPX_Save)()**
- **(*xcpb->Get_Buffer)()**
- **(*xcpb->getcookie)()**
- **(*xcpb->GetFirstRect)()**
- **(*xcpb->GetNextRect)()**
- **(*xcpb->MFsave)()**
- **(*xcpb->Popup)()**
- **(*xcpb->rsh_fix)()**
- **(*xcpb->rsh_obfix)()**
- **(*xcpb->Set_Evnt_Mask)()**
- **(*xcpb->Sl_arrow)()**
- **(*xcpb->Sl_dragx)()**
- **(*xcpb->Sl_dragy)()**
- **(*xcpb->Sl_size)()**
- **(*xcpb->Sl_x)()**
- **(*xcpb->Sl_y)()**
- **(*xcpb->Xform_do)()**
- **(*xcpb->XGen_Alert)()**

(*xcpb->CPX_Save)()

BOOLEAN (*xcpb->CPX_Save)(***ptr*** **,** ***num*** **);**
VOIDP ***ptr*****;**
LONG ***num*****;**

CPX_Save() writes the specified data to the CPX on disk at the beginning of the CPX's DATA segment.

PARAMETERS *ptr* is a pointer to the data to save. *num* specifies the length of the data in bytes.

BINDING

```
(*xcpb->CPX_Save)( ptr, num );
```

RETURN VALUE **CPX_Save()** returns **TRUE** if the operation was successful or **FALSE** if an error occurred.

COMMENTS **CPX_Save()** stores the specified data on disk in the original CPX file at the start of the DATA segment of the program. For this reason, enough space should be allocated to account for this data. See **cpx_init()** for an example method of accomplishing this.

SEE ALSO **(*xcpb->Get_Buffer)()**

(*xcpb->Get_Buffer)()

VOIDP (*xcpb->Get_Buffer)(VOID)

Get_Buffer() returns the address of a 64-byte static storage location for the calling CPX.

BINDING

```
bufptr = (*xcpb->Get_Buffer)();
```

RETURN VALUE **Get_Buffer()** returns a pointer to a 64-byte static storage location which can be used by the CPX to preserve data between invocations.

COMMENTS Data stored in this area is lost upon a reboot. Permanent data should be stored using **CPX_Save()**.

SEE ALSO **(*xcpb->CPX_Save)()**

(*xcpb->getcookie)()

WORD (*xcpb->getcookie)(*cookie, pvalue*)
LONG *cookie*;
LONG **pvalue*;

getcookie() searches the 'cookie jar' for a given cookie and if found returns its stored longword.

PARAMETERS *cookie* contains the longword cookie (usually a packed 4 character ASCII code) to search for. If found, the value of the cookie is placed in the **LONG** pointed to by *pvalue*.

BINDING

```
err = (*xcpb->getcookie)( cookie, pvalue );
```

RETURN VALUE **getcookie()** returns **TRUE** if the value placed in *pvalue* is valid or **FALSE** if the cookie was not found.

COMMENTS This function is useful in locating TSR's or other resident processes which a CPX is designed to configure.

(*xcpb->GetFirstRect)()

GRECT *(*xcpb->GetFirstRect)(*prect*)
GRECT **prect*;

GetFirstRect() returns the first member of the Control Panel's rectangle list intersected by *prect*.

PARAMETERS *prect* points to a **GRECT** containing the extent of the dirtied area.

BINDING

```
rdraw = (*xcpb->GetFirstRect)( prect );
```

RETURN VALUE **GetFirstRect()** will return a pointer to a **GRECT** containing the first intersecting rectangle to redraw or **NULL** if none of the CPX's rectangles intersect the dirtied area.

COMMENTS **Xform_do()** handles resource object redraws in Form CPX's. Other objects requiring a redraw in Form CPX's and all objects in Event CPX's must be redrawn with using these functions when a redraw message is generated.

SEE ALSO **(*xcpb->GetNextRect)()**

(*xcpb->GetNextRect)()

GRECT *(*xcpb->GetNextRect)(VOID)

GetNextRect() returns subsequent rectangles needing to be redrawn after first calling **GetFirstRect()**.

BINDING

```
rdraw = (*xcpb->GetNextRect)();
```

RETURN VALUE **GetNextRect()** returns a pointer to a **GRECT** structure containing a subsequent rectangle needing to be redrawn.

COMMENTS When a redraw message is received, it should be handled as illustrated below (the example given is for an Event CPX but it may be applied to the **WM_REDRAW** message handling section of a Form CPX as well):

```
VOID
cpx_draw( clip )
GRECT *clip;
{
        GRECT *rdraw;

        rdraw = (*xcpb->GetFirstRect)( clip );

        while( rdraw )
        {
          /* User redraw function */
          my_redraw( rdraw );
          rdraw = (*xcpb->GetNextRect)();
        }
}
```

SEE ALSO **(*xcpb->GetFirstRect)()**

(*xcpb->MFsave)()

VOID (*xcpb->MFsave)(***flag, mf*** **)**
BOOLEAN ***flag*;**
MFORM ***mf*;**

MFsave() saves the current mouse form so that a custom application mouse form is not destroyed when the CPX calls **graf_mouse()** or **vsc_form()** to change the shape of the mouse.

PARAMETERS *flag* specifies the action to take. If *flag* is **MFSAVE** (1), the current mouse form will be written into the **MFORM** structure pointed to by *mf*. If *flag* is **MFRESTORE** (0), the mouse form will be restored from the **MFORM** structure pointed to by *mf*. See **vsc_form()** for the definition of **MFORM**.

BINDING

```
(*xcpb->MFsave)( flag, mf );
```

(*xcpb->Popup)()

WORD (*xcpb->Popup)(*items, num_items, default, font, button, world*);
CHAR ****items*[];**
WORD ***num_items, default, font*;**
GRECT ****button*, ***world*;

Popup() displays and controls user interaction with a popup menu.

PARAMETERS

items points to an array of character pointers pointing to the text of the items. Each string must be padded in front with at least 2 spaces and should be of equal length (at least as long as the longest string). *num_items* specifies the number of items to display in the popup. If *num_items* exceeds five, the popup will only show three items with two arrows to allow scrolling.

default indicates the default item (the default item is displayed with a checkmark) or -1 to indicate no default item.

font specifies the font size (3 = large, 5 = small) of the items in the popup.

button points to a **GRECT** containing the rectangular extent of the button pressed to call the popup. *world* points to a **GRECT** containing the current extent of the CPX work area.

BINDING

```
ret = (*xcpb->Popup)( items, num_items, default, font, button,
       world );
```

RETURN VALUE

Popup() returns the item selected (0 based) or -1 if no selection was made (the user clicked outside of the popup area).

COMMENTS

This function is unique to CPX's and is not the same as **menu_popup()**.

Button objects which are to be used as popups should be **TOUCHEXIT** objects. In addition, as a matter of style, popup buttons should be **SHADOWED**.

(*xcpb->rsh_fix)()

VOID (*xcpb->rsh_fix)(*num_objs*, *num_frstr*, *num_frimg*, *num_tree*, *rs_object*, *rs_tedinfo*,
**   *rs_strings*, *rs_iconblk*, *rs_bitblk*, *rs_frstr*, *rs_frimg*, *rs_trindex*, *rs_imdope*);**
WORD *num_objs*, *num_frstr*, *num_frimg*, *num_tree*;
OBJECT **rs_object*;
TEDINFO **rs_tedinfo*;
char **rs_strings*[];
ICONBLK **rs_iconblk*;
BITBLK **rs_bitblk*;
LONG **rs_frstr*, **rs_frimg*, **rs_trindex*;
struct foobar **rs_imdope*;

rsh_fix() fixes up a resource tree in memory based on an 8x16 character font.

PARAMETERS

When using the Atari Resource Construction Set the parameters are generated in the .RSH file created by the compiler.

When using other resource construction sets you should refer to their instructions for applying their resource structure to this function or use the CPX function **rsh_obfix()** on each **OBJECT**.

BINDING

```
(xcpb->rsh_fix)( num_objs, num_frstr, num_frimg, num_tree,
       rs_object, rs_tedinfo, rs_strings, rs_iconblk, rs_bitblk,
       rs_frstr, rs_frimg, rs_trindex, rs_imdope );
```

COMMENTS

rsrc_load(), **rsrc_obfix()**, and **rsrc_rcfix()** fix up a resource file based upon the current screen character size. CPX resource data is always fixed up based upon an 8x16 character font.

Resources should be designed on a screen that supports an 8x16 ratio. When using the Atari Resource Construction Set, the resouce should be designed as a ‘Panel’ rather than a ‘Dialog’. With other resource construction applications the same effect is acheived by turning snap off.

Resources should only be fixed up when the *xcpb->SkipRshFix* flag is 0. This prevents resources from being fixed up more than once.

SEE ALSO

(*xcpb->rsh_obfix)()

(*xcpb->rsh_obfix)()

VOID (*xcpb->rsh_obfix)(*tree*, *curob*)
OBJECT **tree*;
WORD *curob*;

rsh_obfix() converts the specified object from character to pixel based coordinates based on an 8x16 character font.

PARAMETERS *tree* points to the **OBJECT** tree which contains the object *curob* to fix up.

BINDING

```
(*xcpb->rsh_obfix)( tree, curob );
```

COMMENTS See **rsh_fix()**.

SEE ALSO **(*xcpb->rsh_fix)()**

(*xcpb->Set_Evnt_Mask)()

VOID (*xcpb->Set_Evnt_Mask)(*mask*, *m1*, *m2*, *time*)
WORD *mask*;
MOBLK **m1*;
MOBLK **m2*;
LONG *time*;

Set_Evnt_Mask() defines which events an Event CPX will process with its callback functions.

PARAMETERS *mask* is a bit mask of events (**MU_MESAG**, **MU_TIMER**, etc...) that the CPX wishes to process as in **evnt_multi()**. *m1* and *m2* point to **MOBLK** structures which define mouse rectangles to wait for if the CPX wishes to wait for **MU_M1** and/or **MU_M2** events as in **evnt_mouse()**. **MOBLK** is defined as follows:

```
typedef struct
{
       WORD m_out; /* 0 = enter, 1 = exit */
       WORD m_x;
       WORD m_y;
       WORD m_w;
       WORD m_h;
} MOBLK;
```

time specifies the length of time to specify for the **MU_TIMER** event if appropriate.

BINDING

```
(*xcpb->Set_Evnt_Mask)( mask, m1, m2, time );
```

COMMENTS

This function is only valid for Event CPX's.

(*xcpb->Sl_arrow)()

VOID (*xcpb->Sl_arrow)(*tree, base, slider, obj, inc, min, max, numvar, dir, foo*)
OBJECT **tree*;
WORD *base, slider, obj, inc, min, max*;
WORD **numvar*;
WORD *dir*;
VOID (foo*)();**

Sl_arrow() is called by a CPX when the user clicks on an arrow element of an 'active' slider.

PARAMETERS

tree points to the object tree containing the slider elements. *base* is the object index of the slider 'track'. *slider* is the object index of the slider 'elevator'. *obj* is the index of the arrow element clicked on by the user.

inc specifies the increment amount for each slider step (+/-). *min* specifies the minimum value the slider can represent. *max* specifies the maximum value the slider can represent.

numvar points to a **WORD** containing the value which the slider represents and which is to be updated as the slider is moved. *dir* specifies the direction of the slider movement (**VERTICAL** (0) or **HORIZONTAL** (1)).

foo is a pointer to a user-defined callback function which is called once for each step of the slider to allow the user's action to 'actively' update the slider. *foo* may be **NULL** if no updating is desired.

BINDING

```
(*xcpb->Sl_arrow)( tree, base, slider, obj, inc, min, max,
       numvar, dir, foo );
```

COMMENTS

Slider paging can be accomplished with this function. To do so use a method similar to the following (this example is for vertical sliders):

```
graf_mkstate( &mx, &my, &dum, &dum );
objc_offset( tree, slider, &ox, &oy );
inc = ( ( my < oy ) ? ( -1 ) : ( 1 ) );
(*xcpb->Sl_arrow( tree, base, slider, base, inc, min, max,
       &numvar, VERTICAL, foo );
```

(*xcpb->Sl_dragx)()

VOID (*xcpb->Sl_dragx)(*tree, base, slider, min, max, numvar, foo*)
OBJECT **tree*;
WORD *base, slider, min, max*;
WORD **numvar*;
VOID (foo*)();**

Sl_dragx() is called by a CPX when a user clicks on the horizontal slider 'elevator' of an 'active' slider.

PARAMETERS *tree* points to an **OBJECT** tree containing the slider elements. *base* is the object index of the slider 'track'. *slider* is the object index of the slider 'elevator'.

min specifies the minimum value the slider can represent. *max* specifies the maximum value the slider can represent.

numvar points to a **WORD** containing the value which the slider represents and which is to be updated as the slider is moved.

foo points to a user-defined routine which is called each time the slider value *numvar* is modified. *foo* may be **NULL** if no updating is desired.

BINDING
```
(*xcpb->Sl_dragx)( tree, base, slider, min, max, numvar, foo );
```

COMMENTS It is appropriate to change the shape of the mouse to **FLAT_HAND** while the user is dragging a slider.

SEE ALSO **(*xcpb->Sl_dragy)()**

(*xcpb->Sl_dragy)()

VOID (*xcpb->Sl_dragx)(*tree, base, slider, min, max, numvar, foo*)
OBJECT **tree*;
WORD *base, slider, min, max*;
WORD **numvar*;
VOID (foo*)();**

Sl_dragy() is called by a CPX when a user clicks on the vertical slider 'elevator' of an 'active' slider.

PARAMETERS See **Sl_dragx()**.

BINDING

```
(*xcpb->Sl_dragy)( tree, base, slider, min, max, numvar, foo );
```

COMMENTS It is appropriate to change the shape of the mouse to **FLAT_HAND** while the user is dragging a slider.

SEE ALSO **(*xcpb->Sl_dragx)()**

(*xcpb->Sl_size)()

VOID (*xcpb->Sl_size)(***tree, base, slider, num_items, visible, direction, min_size*** **)**
OBJECT *****tree***;**
WORD ***base, slider, num_items, visible, direction, min_size*** **;**

Sl_size() adjusts the size of the slider 'track' relative to the size of the slider 'elevator'.

PARAMETERS *tree* points to the **OBJECT** tree containing the slider elements. *base* is the object index of the slider 'track'. *slider* is the object index of the slider 'elevator'.

num_items is the total number of items represented by the slider. *visible* is the number of items actually seen by the user.

direction specifies the direction of the slider as either **VERTICAL** (0) or **HORIZONTAL** (1). *min_size* represents the minimum pixel size of the adjusted slider elevator.

BINDING

```
(*xcpb->Sl_size)( tree, base, slider, num_items, visible,
       direction, min_size );
```

COMMENTS This function does not redraw the slider.

(*xcpb->Sl_x)()

VOID (*xcpb->Sl_x)(***tree, base, slider, value, min, max, foo*** **)**
OBJECT *****tree***;**
WORD ***base, slider, value, min, max*****;**
VOID (***foo*****)();**

Sl_x() updates the position of a horizontal slider within its base.

PARAMETERS *tree* points to an **OBJECT** tree containing the slider elements. *base* is the object index of the slider 'track'. *slider* is the object index of the slider 'elevator'.

value is the value the slider should represent. *min* and *max* are the minimum and maximum values the slider can represent respectively.

If *foo* is not **NULL**, it points to a user-function which is called to redraw the slider.

BINDING

```
(*xcpb->Sl_x)( tree, base, slider, value, min, max, foo );
```

SEE ALSO **(*xcpb->Sl_y)()**

(*xcpb->Sl_y)()

VOID (*xcpb->Sl_y)(*tree, base, slider, value, min, max, foo*)
OBJECT **tree*;
WORD *base, slider, value, min, max*;
VOID (foo*)();**

Sl_y() updates the position of a vertical slider within its base.

PARAMETERS See **Sl_x()**.

BINDING

```
(*xcpb->Sl_y)( tree, base, slider, value, min, max, foo );
```

SEE ALSO **(*xcpb->Sl_x)()**

(*xcpb->Xform_do)()

WORD (*xcpb->Xform_do)(*tree, editobj, msg*)
OBJECT **tree*;
WORD *editobj*;
WORD **msg*;

Xform_do() is a specialized version of **form_do()** designed to handle a CPX object tree and window messages concurrently.

PARAMETERS *tree* should point to an **OBJECT** tree containing a form with the root object being 256x176. *editobj* specifies the editable text object to initially display the text cursor at (or 0 if no editable object exists on the form).

msg should point to an 8 **WORD** array used by the function to store special messages returned by **evnt_multi()**.

BINDING

```
ret = (*xcpb->Xform_do)( tree, editobj, msg );
```

RETURN VALUE

Xform_do() returns the positive object number of the **EXIT** or **TOUCHEXIT** object selected. The high bit of this value indicates if the object was double-clicked and should therefore be masked off if unused. If **Xform_do()** returns a -1, then a message should be processed as contained in *msg*. The structure of messages are the same as in **evnt_multi()**. Possible messages are:

WM_REDRAW
AC_CLOSE
WM_CLOSE
CT_KEY

CT_KEY (53) is a special **XCONTROL** message indicating that a key was pressed. The scancode of the key pressed is contained in *msg[3]*. Only special keyboard keys such as HELP, F1–F10, UNDO, ALT-X, etc... will be returned as the standard alphabetic keys are processed in editable fields.

COMMENTS

The **Xform_do()** function automatically handles and redraws of the given **OBJECT** tree. Any other items needing to be redrawn should be handled at the appropriate window redraw message.

WM_CLOSED messages should always be treated as 'OK' while **AC_CLOSE** messages should be treated as 'Cancel'.

(*xcpb->XGen_Alert)()

BOOLEAN (*xcpb->XGen_Alert)(***id*** **)**
WORD ***id*****;**

XGen_Alert() displays a specialized alert centered in the Control Panel's work area.

PARAMETERS

id specifies the alert to display as follows:

Name	*id*	Alert
SAVE_DEFAULTS	0	Save Defaults?
MEM_ERR	1	Memory Allocation Error
FILE_ERR	2	File I/O Error
FILE_NOT_FOUND	3	File Not Found Error

BINDING

```
ret = (*xcpb->XGen_Alert)( id );
```

RETURN VALUE

XGen_Alert() returns **TRUE** if 'OK' was selected or **FALSE** if 'Cancel' was selected. Alerts 1-3 always returns **TRUE**.