X11 Input Extension Porting Document

X Version 11, Release 5

George Sachs Hewlett-Packard

This document is intended to aid the process of integrating the X11 Input Extension into an X server.

Most of the functionality provided by the input extension is device- and implementation-independent, and should require no changes. The functionality is implemented by routines that typically reside in the server source tree directory extensions/server/xinput. This extension includes functions to enable and disable input extension devices, select input, grab and focus those device, query and change key and button mappings, and others. The only input extension requirements for the device-dependent part of X are that the input devices be correctly initialized and input events from those devices be correctly generated. Device-dependent X is responsible for reading input data from the input device hardware and if necessary, reformatting it into X events.

The process of initializing input extension devices is similar to that used for the core devices, and is described in the following sections. When multiple input devices are attached to X server, the choice of which devices to initially use as the core X pointer and keyboard is left implementation-dependent. It is also up to each implementation to decide whether all input devices will be opened by the server during its initialization and kept open for the life of the server. The alternative is to open only the X keyboard and X pointer during server initialization, and open other input devices only when requested by a client to do so. Either type of implementation is supported by the input extension.

Input extension events generated by the X server use the same 32-byte xEvent wire event as do core input events. However, additional information must be sent for input extension devices, requiring that multiple xEvents be generated each time data is received from an input extension device. These xEvents are combined into a single client XEvent by the input extension library. A later section of this document describes the format and generation of input extension events.

2. Initializing Extension Devices

Extension input devices are initialized in the same manner as the core X input devices. Device-Independent X provides functions that can be called from DDX to initialize these devices. Which functions are called and when will vary by implementation, and will depend on whether the implementation opens all the input devices available to X when X is initialized, or waits until a client requests that a device be opened. In the simplest case, DDX will open all input devices as part of its initialization, when the InitInput routine is called.

2.1. Summary of Calling Sequence

```
Device-Independent X      | Device-Dependent X
--------------------      | -------------------
                          |
InitInput --------------> | - do device-specific initialization
                          |
                          | - call AddInputDevice  (deviceProc,AutoStart)
AddInputDevice            |
  - creates DeviceIntRec  |
  - records deviceProc    |
  - adds new device to    |
    list of off_devices.  |
sets dev->startup=AutoStart|
                          | - call one of:
                          |   - RegisterPointerDevice (X pointer)
                          |     - processInputProc = ProcessPointerEvents
                          |   - RegisterKeyboardDevice (X keyboard)
                          |     - processInputProc = ProcessKeyboardEvents
                          |   - RegisterOtherDevice  (extension device)
                          |     - processInputProc = ProcessOtherEvents
                          |
                          |
InitAndStartDevices -----> | - calls deviceProc with parameters
                          |   (DEVICE_INIT, AutoStart)
sets dev->inited = return |
  value from deviceProc   |
                          |
                          | - in deviceProc, do one of:
                          |   - call InitPointerDeviceStruct (X pointer)
                          |   - call InitKeyboardDeviceStruct (X keybd)
                          |   - init extension device by calling some of:
                          |     - InitKeyClassDeviceStruct
                          |     - InitButtonClassDeviceStruct
                          |     - InitValuatorClassDeviceStruct
                          |     - InitValuatorAxisStruct
                          |     - InitFocusClassDeviceStruct
                          |     - InitProximityClassDeviceStruct
                          |     - InitKbdFeedbackClassDeviceStruct
                          |     - InitPtrFeedbackClassDeviceStruct
                          |     - InitLedFeedbackClassDeviceStruct
                          |     - InitStringFeedbackClassDeviceStruct
                          |     - InitIntegerFeedbackClassDeviceStruct
                          |     - InitBellFeedbackClassDeviceStruct
                          |   - init device name and type by:
                          |     - calling MakeAtom with one of the
                          |       predefined names
                          |     - calling AssignTypeAndName
                          |
                          |
for each device added     |
    by AddInputDevice,    |
    InitAndStartDevices   |
    calls EnableDevice if | - EnableDevice calls deviceProc with
    dev->startup &        |   (DEVICE_ON, AutoStart)
    dev->inited           |
```

```
                        |
If deviceProc returns   | - core devices are now enabled, extension
   Success, EnableDevice |    devices are now available to be accessed
   move the device from  |    through the input extension protocol
   inputInfo.off_devices |    requests.
   to inputInfo.devices  |
```

2.2. Initialization Called From InitInput

InitInput is the first DDX input entry point called during X server startup. This routine is responsible for device- and implementation- specific initialization, and for calling AddInputDevice to create and initialize the DeviceIntRec structure for each input device. AddInputDevice is passed the address of a procedure to be called by the DIX routine InitAndStartDevices when input devices are enabled. This procedure is expected to perform X initialization for the input device.

If the device is to be used as the X pointer, DDX should then call RegisterPointerDevice, passing the DeviceIntRec pointer, to initialize the device as the X pointer.

If the device is to be used as the X keyboard, DDX should instead call RegisterKeyboardDevice to initialize the device as the X keyboard.

If the device is to be used as an extension device, DDX should instead call RegisterOtherDevice, passing the DeviceIntPtr returned by AddInputDevice.

A sample InitInput implementation is shown below.

```
InitInput(argc,argv)
  {
  int i, numdevs, ReadInput();
  DeviceIntPtr dev;
  LocalDevice localdevs[LOCAL_MAX_DEVS];
  DeviceProc kbdproc, ptrproc, extproc;

  /**************************************************************
   * Open the appropriate input devices, determine which are
   * available, and choose an X pointer and X keyboard device
   * in some implementation-dependent manner.
   ***************************************************************/

  open_input_devices (&numdevs, localdevs);

  /**************************************************************
   * Register a WakeupHandler to handle input when it is generated.
   ***************************************************************/

  RegisterBlockAndWakeupHandlers (NoopDDA, ReadInput, NULL);

  /**************************************************************
   * Register the input devices with DIX.
   ***************************************************************/

  for (i=0; i<numdevs; i++)
    {
    if (localdevs[i].use == IsXKeyboard)
      {
      dev = AddInputDevice (kbdproc, TRUE);
      RegisterKeyboardDevice (dev);
      }
    else if (localdevs[i].use == IsXPointer)
      {
      dev = AddInputDevice (ptrproc, TRUE);
      RegisterPointerDevice (dev);
      }
    else
      {
      dev = AddInputDevice (extproc, FALSE);
      RegisterOtherDevice (dev);
      }
    if (dev == NULL)
      FatalError ("Too many input devices.");
    dev->devicePrivate = (pointer) &localdevs[i];
    }
```

2.3. Initialization Called From InitAndStartDevices

After InitInput has returned, InitAndStartDevices is the DIX routine that is called to enable input devices. It calls the device control routine that was passed to AddInputDevice, with a mode value of DEVICE_INIT. The action taken by the device control routine depends on how the device is to be used. If the device is to be the X pointer, the device control routine should call InitPointerDeviceStruct to initialize it. If the device is to be the X keyboard, the device control routine should call InitKeyboardDeviceStruct. Since input extension devices may support various combinations of keys, buttons, valuators, and feedbacks, each class of input that it supports must be initialized. Entry points are defined by DIX to initialize each of the supported classes of input, and are described in the following sections.

A sample device control routine called from InitAndStartDevices is shown below.

```
Bool extproc (dev, mode)
   DeviceIntPtr dev;
   int mode;
   {
   LocalDevice *localdev = (LocalDevice *) dev->devicePrivate;

   switch (mode)
      {
      case DEVICE_INIT:
         if (strcmp(localdev->name, XI_TABLET) == 0)
            {
            /****************************************************
             * This device reports proximity, has buttons,
             * reports two axes of motion, and can be focused.
             * It also supports the same feedbacks as the X pointer
             * (acceleration and threshold can be set).
             ****************************************************/

            InitButtonClassDeviceStruct (dev, button_count, button_map);
            InitValuatorClassDeviceStruct (dev, localdev->n_axes,);
               motionproc, MOTION_BUF_SIZE, Absolute);
            for (i=0; i<localdev->n_axes; i++)
               InitValuatorAxisStruct (dev, i, min_val, max_val,
                  resolution);
            InitFocusClassDeviceStruct (dev);
            InitProximityClassDeviceStruct (dev);
            InitPtrFeedbackClassDeviceStruct (dev, p_controlproc);
            }
         else if (strcmp(localdev->name, XI_BUTTONBOX) == 0)
            {
            /****************************************************
             * This device has keys and LEDs, and can be focused.
             ****************************************************/

            InitKeyClassDeviceStruct (dev, syms, modmap);
            InitFocusClassDeviceStruct (dev);
            InitLedFeedbackClassDeviceStruct (dev, ledcontrol);
            }
         else if (strcmp(localdev->name, XI_KNOBBOX) == 0)
            {
            /****************************************************
             * This device reports motion.
             * It can be focused.
             ****************************************************/

            InitValuatorClassDeviceStruct (dev, localdev->n_axes,);
               motionproc, MOTION_BUF_SIZE, Absolute);
            for (i=0; i<localdev->n_axes; i++)
               InitValuatorAxisStruct (dev, i, min_val, max_val,
                  resolution);
            InitFocusClassDeviceStruct (dev);
            }
         localdev->atom =
            MakeAtom(localdev->name, strlen(localdev->name), FALSE);
```

```
        AssignTypeAndName (dev, localdev->atom, localdev->name);
        break;
    case DEVICE_ON:
        AddEnabledDevice (localdev->file_ds);
        dev->on = TRUE;
        break;
    case DEVICE_OFF:
        dev->on = FALSE;
        RemoveEnabledDevice (localdev->file_ds);
        break;
    case DEVICE_CLOSE:
        break;
    }
}
```

The device control routine is called with a mode value of DEVICE_ON by the DIX routine EnableDevice, which is called from InitAndStartDevices. When called with this mode, it should call AddEnabledDevice to cause the server to begin checking for available input from this device.

>From InitAndStartDevices, EnableDevice is called for all devices that have the "inited" and "startup" fields in the DeviceIntRec set to TRUE. The "inited" field is set by InitAndStartDevices to the value returned by the deviceproc when called with a mode value of DEVICE_INIT. The "startup" field is set by AddInputDevice to value of the second parameter (autoStart).

When the server is first initialized, it should only be checking for input from the core X keyboard and pointer. One way to accomplish this is to call AddInputDevice for the core X keyboard and pointer with an autoStart value equal to TRUE, while calling AddInputDevice for input extension devices with an autoStart value equal to FALSE. If this is done, EnableDevice will skip all input extension devices during server initialization. In this case, the OpenInputDevice routine should set the "startup" field to TRUE when called for input extension devices. This will cause ProcXOpenInputDevice to call EnableDevice for those devices when a client first does an XOpenDevice request.

2.4. DIX Input Class Initialization Routines

DIX routines are defined to initialize each of the defined input classes. The defined classes are:

- KeyClass - the device has keys.
- ButtonClass - the device has buttons.
- ValuatorClass - the device reports motion data or positional data.
- Proximitylass - the device reports proximity information.
- FocusClass - the device can be focused.
- FeedbackClass - the device supports some kind of feedback

DIX routines are provided to initialize the X pointer and keyboard, as in previous releases of X. During X initialization, InitPointerDeviceStruct is called to initialize the X pointer, and InitKeyboardDeviceStruct is called to initialize the X keyboard. There is no corresponding routine for extension input devices, since they do not all support the same classes of input. Instead, DDX is responsible for the initialization of the input classes supported by extension devices. A description of the routines provided by DIX to perform that initialization follows.

2.4.1. InitKeyClassDeviceStruct

This function is provided to allocate and initialize a KeyClassRec, and should be called for extension devices that have keys. It is passed a pointer to the device, and pointers to arrays of keysyms and modifiers reported by the device. It returns FALSE if the KeyClassRec could not be allocated, or if the maps for the keysyms and and modifiers could not be allocated. Its parameters are:

```
Bool
InitKeyClassDeviceStruct(dev, pKeySyms, pModifiers)
   DeviceIntPtr dev;
   KeySymsPtr pKeySyms;
   CARD8 pModifiers[];
```

The DIX entry point InitKeyboardDeviceStruct calls this routine for the core X keyboard. It must be called explicitly for extension devices that have keys.

2.4.2. InitButtonClassDeviceStruct

This function is provided to allocate and initialize a ButtonClassRec, and should be called for extension devices that have buttons. It is passed a pointer to the device, the number of buttons supported, and a map of the reported button codes. It returns FALSE if the ButtonClassRec could not be allocated. Its parameters are:

```
Bool
InitButtonClassDeviceStruct(dev, numButtons, map)
   register DeviceIntPtr dev;
   int numButtons;
   CARD8 *map;
```

The DIX entry point InitPointerDeviceStruct calls this routine for the core X pointer. It must be called explicitly for extension devices that have buttons.

2.4.3. InitValuatorClassDeviceStruct

This function is provided to allocate and initialize a ValuatorClassRec, and should be called for extension devices that have valuators. It is passed the number of axes of motion reported by the device, the address of the motion history procedure for the device, the size of the motion history buffer, and the mode (Absolute or Relative) of the device. It returns FALSE if the ValuatorClassRec could not be allocated. Its parameters are:

```
Bool
InitValuatorClassDeviceStruct(dev, numAxes, motionProc, numMotionEvents, mode)
   DeviceIntPtr dev;
   int (*motionProc)();
   int numAxes;
   int numMotionEvents;
   int mode;
```

The DIX entry point InitPointerDeviceStruct calls this routine for the core X pointer. It must be called explicitly for extension devices that report motion.

2.4.4. InitValuatorAxisStruct

This function is provided to initialize an XAxisInfoRec, and should be called for core and extension devices that have valuators. The space for the XAxisInfoRec is allocated by the InitValuatorClassDeviceStruct function, but is not initialized.

InitValuatorAxisStruct should be called once for each axis of motion reported by the device. Each invocation should be passed the axis number (starting with 0), the minimum value for that axis, the maximum value for that axis, and the resolution of the device in counts per meter. If the device reports relative motion, 0 should be reported as the minimum and maximum values. InitValuatorAxisStruct has the following parameters:

```
InitValuatorAxisStruct(dev, axnum, minval, maxval, resolution)
    DeviceIntPtr dev;
    int axnum;
    int minval;
    int maxval;
    int resolution;
```

This routine is not called by InitPointerDeviceStruct for the core X pointer. It must be called explicitly for core and extension devices that report motion.

2.4.5. InitFocusClassDeviceStruct

This function is provided to allocate and initialize a FocusClassRec, and should be called for extension devices that can be focused. It is passed a pointer to the device, and returns FALSE if the allocation fails. It has the following parameter:

```
Bool
InitFocusClassDeviceStruct(dev)
    DeviceIntPtr dev;
```

The DIX entry point InitKeyboardDeviceStruct calls this routine for the core X keyboard. It must be called explicitly for extension devices that can be focused. Whether or not a particular device can be focused is left implementation-dependent.

2.4.6. InitProximityClassDeviceStruct

This function is provided to allocate and initialize a ProximityClassRec, and should be called for extension absolute pointing devices that report proximity. It is passed a pointer to the device, and returns FALSE if the allocation fails. It has the following parameter:

```
Bool
InitProximityClassDeviceStruct(dev)
    DeviceIntPtr dev;
```

2.4.7. Initializing Feedbacks

2.4.7.1. InitKbdFeedbackClassDeviceStruct

This function is provided to allocate and initialize a KbdFeedbackClassRec, and may be called for extension devices that support some or all of the feedbacks that the core keyboard supports. It is passed a pointer to the device, a pointer to the procedure that sounds the bell, and a pointer to the device control procedure. It returns FALSE if the allocation fails, and has the following parameters:

```
Bool
InitKbdFeedbackClassDeviceStruct(dev, bellProc, controlProc)
    DeviceIntPtr dev;
    void (*bellProc)();
    void (*controlProc)();
```

The DIX entry point InitKeyboardDeviceStruct calls this routine for the core X keyboard. It must be called explicitly for extension devices that have the same feedbacks as a keyboard. Some feedbacks, such as LEDs and bell, can be supported either with a KbdFeedbackClass or with BellFeedbackClass and LedFeedbackClass feedbacks.

2.4.7.2. InitPtrFeedbackClassDeviceStruct

This function is provided to allocate and initialize a PtrFeedbackClassRec, and should be called for extension devices that allow the setting of acceleration and threshold. It is passed a pointer to the device, and a pointer to the device control procedure. It returns FALSE if the allocation fails, and has the following parameters:

```
Bool
InitPtrFeedbackClassDeviceStruct(dev, controlProc)
   DeviceIntPtr dev;
   void (*controlProc)();
```

The DIX entry point InitPointerDeviceStruct calls this routine for the core X pointer. It must be called explicitly for extension devices that support the setting of acceleration and threshold.

2.4.7.3. InitLedFeedbackClassDeviceStruct

This function is provided to allocate and initialize a LedFeedbackClassRec, and should be called for extension devices that have LEDs. It is passed a pointer to the device, and a pointer to the device control procedure. It returns FALSE if the allocation fails, and has the following parameters:

```
Bool
InitLedFeedbackClassDeviceStruct(dev, controlProc)
   DeviceIntPtr dev;
   void (*controlProc)();
```

Up to 32 LEDs per feedback can be supported, and a device may have multiple feedbacks of the same type.

2.4.7.4. InitBellFeedbackClassDeviceStruct

This function is provided to allocate and initialize a BellFeedbackClassRec, and should be called for extension devices that have a bell. It is passed a pointer to the device, and a pointer to the device control procedure. It returns FALSE if the allocation fails, and has the following parameters:

```
Bool
InitBellFeedbackClassDeviceStruct(dev, bellProc, controlProc)
   DeviceIntPtr dev;
   void (*bellProc)();
   void (*controlProc)();
```

2.4.7.5. InitStringFeedbackClassDeviceStruct

This function is provided to allocate and initialize a StringFeedbackClassRec, and should be called for extension devices that have a display upon which a string can be displayed. It is passed a pointer to the device, and a pointer to the device control procedure. It returns FALSE if the allocation fails, and has the following parameters:

```
Bool
InitStringFeedbackClassDeviceStruct(dev, controlProc, max_symbols,
        num_symbols_supported, symbols)
   DeviceIntPtr dev;
   void (*controlProc)();
   int max_symbols:
   int num_symbols_supported;
   KeySym *symbols;
```

2.4.7.6. InitIntegerFeedbackClassDeviceStruct

This function is provided to allocate and initialize an IntegerFeedbackClassRec, and should be called for extension devices that have a display upon which an integer can be displayed. It is passed a pointer to the device, and a pointer to the device control procedure. It returns FALSE if the allocation fails, and has the following parameters:

```
Bool
InitIntegerFeedbackClassDeviceStruct(dev, controlProc)
    DeviceIntPtr dev;
    void (*controlProc)();
```

2.5. Initializing The Device Name And Type

The device name and type can be initialized by calling AssignTypeAndName with the following parameters:

```
void
AssignTypeAndName(dev, type, name)
    DeviceIntPtr dev;
    Atom type;
    char *name;
```

This will allocate space for the device name and copy the name that was passed. The device type can be obtained by calling MakeAtom with one of the names defined for input devices. MakeAtom has the following parameters:

```
Atom
MakeAtom(name, len, makeit)
    char *name;
    int len;
    Bool makeit;
```

Since the atom was already made when the input extension was initialized, the value of makeit should be FALSE;

3. Closing Extension Devices

The DisableDevice entry point is provided by DIX to disable input devices. It calls the device control routine for the specified device with a mode value of DEVICE_OFF. The device control routine should call RemoveEnabledDevice to stop the server from checking for input from that device.

DisableDevice is not called by any input extension routines. It can be called from the CloseInputDevice routine, which is called by ProcXCloseDevice when a client makes an XCloseDevice request. If DisableDevice is called, it should only be called when the last client using the extension device has terminated or called XCloseDevice.

4. Implementation-Dependent Routines

Several input extension protocol requests have implementation-dependent entry points. Default routines are defined for these entry points and contained in the source file extensions/server/xinput/xstubs.c. Some implementations may be able to use the default routines without change. The following sections describe each of these routines.

4.1. AddOtherInputDevices

AddOtherInputDevice is called from ProcXListInputDevices as a result of an XListInputDevices protocol request. It may be needed by implementations that do not open extension input devices until requested to do so by some client. These implementations may not initialize all devices when the X server starts up, because some of those devices may be in use. Since the XListInputDevices function only lists those devices that have been initialized, AddOtherInputDevices is called to give DDX a chance to initialize any previously unavailable input devices.

A sample AddOtherInputDevices routine might look like the following:

```
void
AddOtherInputDevices ()
    {
    DeviceIntPtr dev;
    int i;

    for (i=0; i<MAX_DEVICES; i++)
        {
        if (!local_dev[i].initialized && available(local_dev[i]))
            {
            dev = (DeviceIntPtr) AddInputDevice (local_dev[i].deviceProc, TRUE);
            dev->public.devicePrivate = local_dev[i];
            RegisterOtherDevice (dev);
            dev->inited = ((*dev->deviceProc)(dev, DEVICE_INIT) == Success);
            }
        }
    }
```

The default AddOtherInputDevices routine in xstubs.c does nothing. If all input extension devices are initialized when the server starts up, it can be left as a null routine.

4.2. OpenInputDevice

Some X server implementations open all input devices when the server is initialized and never close them. Other implementations may open only the X pointer and keyboard devices during server initialization, and open other input devices only when some client makes an XOpenDevice request. This entry point is for the latter type of implementation.

If the physical device is not already open, it can be done in this routine. In this case, the server must keep track of the fact that one or more clients have the device open, and physically close it when the last client that has it open makes an XCloseDevice request.

The default implementation is to do nothing (assume all input devices are opened during X server initialization and kept open).

4.3. CloseInputDevice

Some implementations may close an input device when the last client using that device requests that it be closed, or terminates. CloseInputDevice is called from ProcXCloseDevice when a client makes an XCloseDevice protocol request.

The default implementation is to do nothing (assume all input devices are opened during X server initialization and kept open).

4.4. SetDeviceMode

Some implementations support input devices that can report either absolute positional data or relative motion. The XSetDeviceMode protocol request is provided to allow DDX to change the current mode of such a device.

The default implementation is to always return a BadMatch error. If the implementation does not support any input devices that are capable of reporting both relative motion and absolute position information, the default implementation may be left unchanged.

4.5. SetDeviceValuators

Some implementations support input devices that allow their valuators to be set to an initial value. The XSetDeviceValuators protocol request is provided to allow DDX to set the valuators of such a device.

The default implementation is to always return a BadMatch error. If the implementation does not support any input devices that are allow their valuators to be set, the default implementation may be left unchanged.

4.6. ChangePointerDevice

The XChangePointerDevice protocol request is provided to change which device is used as the X pointer. Some implementations may maintain information specific to the X pointer in the private data structure pointed to by the DeviceIntRec. ChangePointerDevice is called to allow such implementations to move that information to the new pointer device. The current location of the X cursor is an example of the type of information that might be affected.

The DeviceIntRec structure that describes the X pointer device does not contain a FocusRec. If the device that has been made into the new X pointer was previously a device that could be focused, ProcXChangePointerDevice will free the FocusRec associated with that device.

If the server implementation desires to allow clients to focus the old pointer device (which is now accessible through the input extension), it should call InitFocusClassDeviceStruct for the old pointer device.

The XChangePointerDevice protocol request also allows the client to choose which axes of the new pointer device are used to move the X cursor in the X- and Y- directions. If the axes are different than the default ones, the server implementation should record that fact.

If the server implementation supports input devices with valuators that are not allowed to be used as the X pointer, they should be screened out by this routine and a BadDevice error returned.

The default implementation is to do nothing.

4.7. ChangeKeyboardDevice

The XChangeKeyboardDevice protocol request is provided to change which device is used as the X keyboard. Some implementations may maintain information specific to the X keyboard in the private data structure pointed to by the DeviceIntRec. ChangeKeyboardDevice is called to allow such implementations to move that information to the new keyboard device.

The X keyboard device can be focused, and the DeviceIntRec that describes that device has a FocusRec. If the device that has been made into the new X keyboard did not previously have a FocusRec, ProcXChangeKeyboardDevice will allocate one for it.

If the implementation does not want clients to be able to focus the old X keyboard (which has now become available as an input extension device) it should call DeleteFocusClassDeviceStruct to free the FocusRec.

If the implementation supports input devices with keys that are not allowed to be used as the X keyboard, they should be checked for here, and a BadDevice error returned.

The default implementation is to do nothing.

5. Input Extension Events

Events accessed through the input extension are analogous to the core input events, but have different event types. They are of types **DeviceKeyPress**, **DeviceKeyRelease**, **DeviceButtonPress**, **DeviceButtonRelease**, **DeviceDeviceMotionNotify**, **DeviceProximityIn**, **DeviceProximityOut**, and **DeviceValuator**. These event types are not constants. Instead, they are external integers defined by the input extension. Their actual values will depend on which extensions are supported by a server, and the order in which they are initialized.

The data structures that define these events are defined in the file **extensions/include/XIproto.h**. Other input extension constants needed by DDX are defined in the file **extensions/include/XI.h**.

Some events defined by the input extension contain more information than can be contained in the 32-byte xEvent data structure. To send this information to clients, DDX must generate two or more 32-byte wire events. The following sections describe the contents of these events.

5.1. Device Key Events

DeviceKeyPresss events contain all the information that is contained in a core **KeyPress** event, and also the following additional information:

- deviceid - the identifier of the device that generated the event.
- device_state - the state of any modifiers on the device that generated the event
- num_valuators - the number of valuators reported in this event.
- first_valuator - the first valuator reported in this event.
- valuator0 through valuator5 - the values of the valuators.

In order to pass this information to the input extension library, two 32-byte wire events must be generated by DDX. The first has an event type of **DeviceKeyPress**, and the second has an event type of **DeviceValuator**.

The following code fragment shows how the two wire events could be initialized:

```
extern int DeviceKeyPress;
DeviceIntPtr dev;
xEvent xE[2];
CARD8 id, num_valuators;
INT16 x, y, pointerx, pointery;
Time timestamp;
deviceKeyButtonPointer *xev = (deviceKeyButtonPointer *) xE;
deviceValuator *xv;

xev->type = DeviceKeyPress;          /* defined by input extension */
xev->detail = keycode;          /* key pressed on this device */
xev->time = timestamp;          /* same as for core events    */
xev->rootX = pointerx;          /* x location of core pointer */
xev->rootY = pointery;          /* y location of core pointer */

/******************************************************************/
/*                                                   */
/* The following field does not exist for core input events.     */
/* It contains the device id for the device that generated the    */
/* event, and also indicates whether more than one 32-byte wire   */
/* event is being sent.                                   */
/*                                                   */
/******************************************************************/

xev->deviceid = dev->id | MORE_EVENTS;       /* sending more than 1*/

/******************************************************************/
/* Fields in the second 32-byte wire event:                  */
/******************************************************************/

xv = (deviceValuator *) ++xev;
xv->type = DeviceValuator;         /* event type of second event */
xv->deviceid = dev->id;          /* id of this device         */
xv->num_valuators = 0;          /* no valuators being sent    */
xv->device_state  = 0;          /* will be filled in by DIX   */
```

5.2. Device Button Events

DeviceButton events contain all the information that is contained in a core button event, and also the same additional information that a **DeviceKey** event contains.

5.3. Device Motion Events

DeviceMotion events contain all the information that is contained in a core motion event, and also additional valuator information. At least two wire events are required to contain this information. The following code fragment shows how the two wire events could be initialized:

```
extern int DeviceMotionNotify;
DeviceIntPtr dev;
xEvent xE[2];
CARD8 id, num_valuators;
INT16 x, y, pointerx, pointery;
Time timestamp;
deviceKeyButtonPointer *xev = (deviceKeyButtonPointer *) xE;
deviceValuator *xv;

xev->type = DeviceMotionNotify;     /* defined by input extension */
xev->detail = keycode;              /* key pressed on this device */
xev->time = timestamp;              /* same as for core events    */
xev->rootX = pointerx;              /* x location of core pointer */
xev->rootY = pointery;              /* y location of core pointer */

/******************************************************************/
/*                                                                */
/* The following field does not exist for core input events.    */
/* It contains the device id for the device that generated the   */
/* event, and also indicates whether more than one 32-byte wire  */
/* event is being sent.                                           */
/*                                                                */
/******************************************************************/

xev->deviceid = dev->id | MORE_EVENTS;      /* sending more than 1*/

/******************************************************************/
/* Fields in the second 32-byte wire event:                       */
/******************************************************************/

xv = (deviceValuator *) ++xev;
xv->type = DeviceValuator;          /* event type of second event */
xv->deviceid = dev->id;             /* id of this device          */
xv->num_valuators = 2;              /* 2 valuators being sent     */
xv->first_valuator = 0;             /* first valuator being sent  */
xv->device_state  = 0;              /* will be filled in by DIX   */
xv->valuator0 = x;                  /* first axis of this device  */
xv->valuator1 = y;                  /* second axis of this device */
```

Up to six axes can be reported in the deviceValuator event. If the device is reporting more than 6 axes, additional pairs of DeviceMotionNotify and DeviceValuator events should be sent, with the first_valuator field set correctly.

5.4. Device Proximity Events

Some input devices that report absolute positional information, such as graphics tablets and touchscreens, may report proximity events. **ProximityIn** events are generated when a pointing device like a stylus, or in the case of a touchscreen, the user's finger, comes into close proximity with the surface of the input device. **ProximityOut** events are generated when the stylus or finger leaves the proximity of the input devices surface.

Proximity events contain almost the same information as button events. The event type is **ProximityIn** or **ProximityOut**, and there is no detail information.