Libraries

COLLABORATORS			
	TITLE : Libraries		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 23, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 31 Commodities Exchange Library

This chapter describes Commodities Exchange, the library of routines used to add a custom input handler to the Amiga. With Commodities Exchange, any program function can be associated with key combinations or other input events globally allowing the creation utility programs that run in the background for all tasks.

1.2 31 Commodities Exchange Library / Custom Input Handlers

The input.device has a hand in almost all user input on the Amiga. It gathers input events from the keyboard, the gameport (mouse), and several other sources, into one input "stream". Special programs called input event handlers intercept input events along this stream, examining and sometimes changing the input events. Both Intuition and the console device use input handlers to process user input.

Figure 31-1: The Amiga Input Stream

Using the input.device, a program can introduce its own custom handler into the chain of input handlers at almost any point in the chain. "Hot key" programs, shell pop-up programs, and screen blankers all commonly use custom input handlers to monitor user input before it gets to the Intuition input handler.

Figure 31-2: A Custom Input Handler

Custom input handlers do have their drawbacks, however. Not only are these handlers hard to program, but because there is no standard way to implement and control them, multiple handlers often do not work well together. Their antisocial behavior can result in load order dependencies and incompatibilities between different custom input handlers. Even for the expert user, having several custom input handlers coexist peacefully can be next to impossible.

Figure 31-3: The Commodities Network

Commodities Exchange eliminates these problems by providing a simple, standardized way to program and control custom input handlers. It is divided into three parts: an Exec library, a controller program, and some amiga.lib functions.

The Exec library is called commodities.library. When it is first opened, commodities.library establishes a single input handler just before Intuition in the input chain. When this input handler receives an input event, it creates a CxMessage (Commodities Exchange Message) corresponding to the input event, and diverts the CxMessage through the network of Commodities Exchange input handlers (Figure 31-3).

These handlers are made up of trees of different CxObjects (Commodities Exchange Objects), each of which performs a simple operation on the CxMessages. Any CxMessages that exit the network are returned to the input.device's input stream as input events.

Through function calls to the commodities.library, an application can install a custom input handler. A Commodities Exchange application, sometimes simply referred to as a commodity, uses the CxObject primitives to do things such as filter certain CxMessages, translate CxMessages, signal a task when a CxObject receives a CxMessage, send a message when a CxObject receives a CxMessage, or if necessary, call a custom function when a CxObject receives a CxMessage.

The controller program is called Commodities Exchange. The user can monitor and control all the currently running Commodities Exchange applications from this one program. The user can enable and disable a commodity, kill a commodity, or, if the commodity has a window, ask the commodity to show or hide its window. When the user requests any of these actions, the controller program sends the commodity a message, telling it which action to perform.

The third component of Commodities Exchange is its scanned library functions. These functions are part of the amiga.lib scanned library. They do a lot of the work involved with parsing command lines and Tool Types.

Commodities Exchange is ideal for programs like hot keys/pop ups, screen blankers, and mouse blankers that need to monitor all user input. Commodities Exchange should never be used as an alternate method of receiving user input for an application. Other applications depend on getting user input in some form or another from the input stream. A greedy program that diverts input to itself rather than letting the input go to where the user expects it can seriously confuse the user, not to

mention compromise the advantages of multitasking.

1.3 31 Commodities Exchange Library / CxObjects

CxObjects are the basic building blocks used to construct a commodity. A commodity uses CxObjects to take care of all manipulations of CxMessages. When a CxMessage "arrives" at a CxObject, that CxObject carries out its primitive action and then, if it has not deleted the CxMessage, it passes the CxMessage on to the next CxObject. A commodity links together CxObjects into a tree, organizing these simple action objects to perform some higher function.

A CxObject is in one of two states, active or inactive. An active CxObject performs its primitive action every time it receives a CxMessage. If a CxObject is inactive, CxMessages bypass it, continuing to the CxObject that follows the inactive one. By default, all CxObjects except the type called brokers are created in the active state.

Currently, there are seven types of CxObjects (Table 31-1).

Object Type	Purpose
Broker	Registers a new commodity with the commodity network
Filter	Accepts or rejects input events based on criteria set up by the application
Sender	Sends a message to a message port
Translate	Replaces the input event with a different one
Signal	Signals a task
Custom	Calls a custom function provided by the commodity
Debug	Sends debug information out the serial port

Table 31-1: Commodities Exchange Object Types

1.4 31 Commodities Exchange Library / Installing A Broker Object

The Commodities Exchange input handler maintains a master list of CxObjects to which it diverts input events using CxMessages. The CxObjects in this master list are a special type of CxObject called brokers. The only thing a broker CxObject does is divert CxMessages to its own personal list of CxObjects. A commodity creates a broker and attaches other CxObjects to it. These attached objects take care of the actual input handler related work of the commodity and make up the broker's personal list.

The first program listing, Broker.c, is a very simple example of a working commodity. It serves only to illustrate the basics of a commodity, not to actually perform any useful function. It shows how to set up a broker and process commands from the controller program.

Besides opening commodities.library and creating an Exec message port,

setting up a commodity requires creating a broker. The function CxBroker() creates a broker and adds it to the master list.

```
CxObj *CxBroker(struct NewBroker *nb, long *error);
```

CxBroker()'s first argument is a pointer to a NewBroker structure:

```
struct NewBroker {
    BYTE    nb_Version;
            /* There is an implicit pad byte after this BYTE */
    BYTE    *nb_Name;
    BYTE    *nb_Title;
    BYTE    *nb_Descr;
    SHORT   nb_Unique;
    SHORT   nb_Flags;
    BYTE    nb_Pri;
            /* There is an implicit pad byte after this BYTE */
    struct  MsgPort   *nb_Port;
    WORD    nb_ReservedChannel;
            /* Unused, make zero for future compatibility */
};
```

Commodities Exchange gets all the information it needs about the broker from this structure. NewBroker's nb_Version field contains the version number of the NewBroker structure. This should be set to NB_VERSION which is defined in <libraries/commodities.h>. The nb_Name, nb_Title, and nb_Descr point to strings which hold the name, title, and description of the broker. The two bit fields, nb_Unique and nb_Flags, toggle certain features of Commodities Exchange based on their values. They are discussed in detail later in this chapter.

The nb_Pri field contains the broker's priority. Commodities Exchange inserts the broker into the master list based on this number. Higher priority brokers get CxMessages before lower priority brokers.

CxBroker()'s second argument is a pointer to a LONG. If this pointer is not NULL, CxBroker() fills in this field with one of the following error return codes from <libraries/commodities.h>:

```
CBERR_OK        0       /* No error                         */
CBERR_SYSERR    1       /* System error , no memory, etc    */
CBERR_DUP       2       /* uniqueness violation             */
CBERR_VERSION   3       /* didn't understand nb_VERSION     */
```

Once the broker object is created with CxBroker(), it must be activated with ActivateCxObj().

```
oldactivationvalue = LONG ActivateCxObj(CxObj *co,
                                        long newactivationvalue);
```

After successfully completing the initial set up and activating the broker, a commodity can begin its input processing loop waiting for CxMessages to arrive.

1.5 31 Commodities Exchange Library / CxMessages

There are actually two types of CxMessages. The first, CXM_IEVENT, corresponds to an input event and travels through the Commodities Exchange network. The other type, CXM_COMMAND, carries a command to a commodity. A CXM_COMMAND normally comes from the controller program and is used to pass user commands on to a commodity. A commodity receives these commands through an Exec message port that the commodity sets up before it calls CxBroker(). The NewBroker's nb_Port field points to this message port. A commodity can tell the difference between the two types of CxMessages by calling the CxMsgType() function.

```
ULONG  CxMsgType( CxMsg *cxm );
UBYTE *CxMsgData( CxMsg *cxm );
LONG   CxMsgID  ( CxMsg *cxm );
```

A CxMessage not only has a type, it can also have a data pointer as well as an ID associated with it. The data associated with a CXM_IEVENT CxMessage is an InputEvent structure. By using the CxMsgData() function, a commodity can obtain a pointer to the corresponding InputEvent of a CXM_IEVENT message. Commodities Exchange gives an ID of zero to any CXM_IEVENT CxMessage that it introduces to the Commodities network but certain CxObjects can assign an ID to them.

For a CXM_COMMAND CxMessages, the data pointer is generally not used but the ID specifies a command passed to the commodity from the user operating the controller program. The CxMsgID() macro extracts the ID from a CxMessage.

1.6 31 / CxMessages / Controller Commands

The commands that a commodity can receive from the controller program (as defined in <libraries/commodities.h>) are:

```
CXCMD_DISABLE     /* please disable yourself       */
CXCMD_ENABLE      /* please enable yourself        */
CXCMD_KILL        /* go away for good              */
CXCMD_APPEAR      /* open your window, if you can  */
CXCMD_DISAPPEAR   /* hide your window              */
```

The CXCMD_DISABLE, CXCMD_ENABLE, and CXCMD_KILL commands correspond to the similarly named controller program gadgets, Disable, Enable, and Kill; CXCMD_APPEAR and CXCMD_DISAPPEAR correspond to the controller program gadgets, Show and Hide. These gadgets are ghosted in Broker.c because it has no window (It doesn't make much sense to give the user a chance to click the Show and Hide gadgets). In order to do this, Broker.c has to tell Commodities Exchange to ghost these gadgets. When CxBroker() sets up a broker, it looks at the NewBroker.nb_Flags field to see if the COF_SHOW_HIDE bit (from <libraries/commodities.h>) is set. If it is, the "Show" and "Hide" gadgets for this broker will be selectable. Otherwise

they are ghosted and disabled.

1.7 31 / CxMessages / Shutting Down the Commodity

Shutting down a commodity is easy. After replying to all CxMessages waiting at the broker's message port, a commodity can delete its CxObjects. The DeleteCxObj() function removes a single CxObject from the Commodities network. DeleteCxObjAll() removes multiple objects.

```
void DeleteCxObj( CxObj *co );
void DeleteCxObjAll( CxObj *delete_co );
```

If a commodity has a lot of CxObjects, deleting each individually can be a bit tedious. DeleteCxObjAll() will delete a CxObject and any other CxObjects that are attached to it. The HotKey.c example given later in this chapter uses this function to delete all its CxObjects. A commodity that uses DeleteCxObjAll() to delete all its CxObjects should make sure that they are all connected to the main one. (See the section "Connecting CxObjects" below.)

After deleting its CxObjects, a commodity must take care of any CxMessages that might have arrived at the message port just before the commodity deleted its objects.

```
while(msg = (CxMsg *)GetMsg(broker_mp))
    ReplyMsg((struct Message *)msg);
```

1.8 31 Commodities Exchange Library / Commodity Tool Types

A goal of Commodities Exchange is to improve user control over input handlers. One way in which it accomplishes this goal is through the use of standard icon Tool Types. The user will expect commodities to recognize the set of standard Tool Types:

```
CX_PRIORITY
CX_POPUP
CX_POPKEY
```

CX_PRIORITY lets the user set the priority of a commodity. The string "CX_PRIORITY=" is a number from -128 to 127. The higher the number, the higher the priority of the commodity, giving it access to input events before lower priority commodities. All commodities should recognize CX_PRIORITY.

CX_POPUP and CX_POPKEY are only relevant to commodities with a window. The string "CX_POPUP=" should be followed by a "yes" or "no", telling the commodity if it should or shouldn't show its window when it is first launched. CX_POPKEY is followed by a string describing the key to use as a hot key for making the commodity's window appear (pop up). The description string for CX_POPKEY describes an input event. The specific format of the string is discussed in the next section ("Filter Objects and the Input Description String").

Commodities Exchange's support library functions simplify parsing arguments from either the Workbench or the Shell (CLI). A Workbench launched commodity gets its arguments directly from the Tool Types in the commodity's icon. Shell launched commodities get their arguments from the command line, but these arguments look exactly like the Tool Types from the commodity's icon. For example, the following command line sets the priority of a commodity called HotKey to 5:

```
HotKey "CX_PRIORITY=5"
```

Commodities Exchange has several support library functions used to parse arguments:

```
tooltypearray = UBYTE **ArgArrayInit(LONG argc, UBYTE **argv);
                void    ArgArrayDone(void);

tooltypevalue = STRPTR  ArgString(UBYTE **tooltypearray,
                                  STRPTR tooltype,
                                  STRPTR defaultvalue);
tooltypevalue = LONG  *ArgInt(UBYTE **tooltypearray,
                              STRPTR tooltype,
                              LONG defaultvalue);
```

ArgArrayInit() initializes a Tool Type array of strings which it creates from the startup arguments, argc and argv. It doesn't matter if these startup arguments come from the Workbench or from a Shell, ArgArrayInit() can extract arguments from either source. Because ArgArrayInit() uses some icon.library functions, a commodity is responsible for opening that library before using the function.

ArgArrayInit() also uses some resources that must be returned to the system when the commodity is done. ArgArrayDone() performs this clean up. Like ArgArrayInit(), ArgArrayDone() uses icon.library, so the library has to remain open until ArgArrayDone() is finished.

The support library has two functions that use the Tool Type array set up by ArgArrayInit(), ArgString() and ArgInt(). ArgString() scans the Tool Type array for a specific Tool Type. If successful, it returns a pointer to the value associated with that Tool Type. If it doesn't find the Tool Type, it returns the default value passed to it. ArgInt() is similar to ArgString(). It also scans the ArgArrayInit()'s Tool Type array, but it returns a LONG rather than a string pointer. ArgInt() extracts the integer value associated with a Tool Type, or, if that Tool Type is not present, it returns the default value.

Of course, these Tool Type parsing functions are not restricted to the standard Commodities Exchange Tool Types. A commodity that requires any arguments should use these functions along with custom Tool Types to obtain these values. Because the Commodities Exchange standard arguments are processed as Tool Types, the user will expect to enter other arguments as Tool Types too.

1.9 31 Commodities Exchange / Filter Objects and Input Description Strings

Because not all commodities are interested in every input event that makes it way down the input chain, Commodities Exchange has a method for filtering them. A filter CxObject compares the CxMessages it receives to a pattern. If a CxMessage matches the pattern, the filter diverts the CxMessage down its personal list of CxObjects.

```
CxObj *CxFilter(UBYTE *descriptionstring);
```

The C macro CxFilter() (defined in <libraries/commodities.h>) returns a pointer to a filter CxObject. The macro has only one argument, a pointer to a string describing which input events to filter. The following regular expression outlines the format of the input event description string (CX_POPKEY uses the same description string format):

```
[class]  {[-] (qualifier|synonym)}  [[-] upstroke]  [highmap|ANSICode]
```

Class can be any one of the class strings in the table below. Each class string corresponds to a class of input event as defined in <devices/inputevent.h>. Commodities Exchange will assume the class is rawkey if the class is not explicitly stated.

Class String	Input Event Class
"rawkey"	IECLASS_RAWKEY
"rawmouse"	IECLASS_RAWMOUSE
"event"	IECLASS_EVENT
"pointerpos"	IECLASS_POINTERPOS
"timer"	IECLASS_TIMER
"newprefs"	IECLASS_NEWPREFS
"diskremoved"	IECLASS_DISKREMOVED
"diskinserted"	IECLASS_DISKINSERTED

Qualifier is one of the qualifier strings from the table below. Each string corresponds to an input event qualifier as defined in <devices/inputevent.h>). A dash preceding the qualifier string tells the filter object not to care if that qualifier is present in the input event. Notice that there can be more than one qualifier (or none at all) in the input description string.

Qualifier String	Input Event Class
"lshift"	IEQUALIFIER_LSHIFT
"rshift"	IEQUALIFIER_RSHIFT
"capslock"	IEQUALIFIER_CAPSLOCK
"control"	IEQUALIFIER_CONTROL
"lalt"	IEQUALIFIER_LALT
"ralt"	IEQUALIFIER_RALT
"lcommand"	IEQUALIFIER_LCOMMAND
"rcommand"	IEQUALIFIER_RCOMMAND
"numericpad"	IEQUALIFIER_NUMERICPAD
"repeat"	IEQUALIFIER_REPEAT
"midbutton"	IEQUALIFIER_MIDBUTTON
"rbutton"	IEQUALIFIER_RBUTTON

```
    "leftbutton"        IEQUALIFIER_LEFTBUTTON
    "relativemouse"     IEQUALIFIER_RELATIVEMOUSE
```

Synonym is one of the synonym strings from the table below. These strings act as synonyms for groups of qualifiers. Each string corresponds to a synonym identifier as defined in <libraries/commodities.h>. A dash preceding the synonym string tells the filter object not to care if that synonym is present in the input event. Notice that there can be more than one synonym (or none at all) in the input description string.

```
    Synonym      Synonym
    String      Identifier
    -------     ----------
    "shift"     IXSYM_SHIFT     /* look for either shift key */
    "caps"      IXSYM_CAPS      /* look for either shift key or capslock */
    "alt"       IXSYM_ALT       /* look for either alt key */
```

Upstroke is the literal string "upstroke". If this string is absent, the filter considers only downstrokes. If it is present alone, the filter considers only upstrokes. If preceded by a dash, the filter considers both upstrokes and downstrokes.

Highmap is one of the following strings:

```
    "space", "backspace", "tab", "enter", "return", "esc", "del",
    "up", "down", "right", "left", "f1", "f2", "f3", "f4", "f5",
    "f6", "f7", "f8", "f9", "f10", "help".
```

ANSICode is a single character (for example ‘a’) that Commodities Exchange looks up in the system default keymap.

Here are some example description strings. For function key F2 with the left Shift and either Alt key pressed, the input description string would be:

```
    "rawkey lshift alt f2"
```

To specify the key that produces an ‘a’ (this may or may not be the A key depending on the keymap), with or without any Shift, Alt, or control keys pressed use:

```
    "-shift -alt -control a"
```

For a mouse move with the right mouse button down, use:

```
    "rawmouse rbutton"
```

To specify a timer event use:

```
    "timer"
```

1.10 31 Commodities Exchange Library / Connecting CxObjects

A CxObject has to be inserted into the Commodities network before it can process any CxMessages. AttachCxObj() adds a CxObject to the personal list of another CxObject. The HotKey.c example uses it to attach its filter to a broker.

```
    void AttachCxObj ( CxObj *headobj, CxObj *co);
    void InsertCxObj ( CxObj *headobj, CxObj *co, CxObj *co_pred );
    void EnqueueCxObj( CxObj *headobj, CxObj *co );
    void SetCxObjPri ( CxObj *co, long pri );
    void RemoveCxObj ( CxObj *co );
```

AttachCxObj() adds the CxObject to the end of headobj's personal list. The ordering of a CxObject list determines which object gets CxMessages first. InsertCxObj() also inserts a CxObject, but it inserts it after another CxObject already in the personal list (co_pred in the prototype above).

Brokers aren't the only CxObjects with a priority. All CxObjects have a priority associated with them. To change the priority of any CxObject, use the SetCxObjPri() function. A commodity can use the priority to keep CxObjects in a personal list sorted by their priority. The commodities.library function EnqueueCxObj() inserts a CxObject into another CxObject's personal list based on priority.

Like its name implies, the RemoveCxObj() function removes a CxObject from a personal list. Note that it is not necessary to remove a CxObject from a list in order to delete it.

```
     HotKey.c
```

1.11 31 Commodities Exchange Library / Sender CxObjects

A filter CxObject by itself is not especially useful. It needs some other CxObjects attached to it. A commodity interested in knowing if a specific key was pressed uses a filter to detect and divert the corresponding CxMessage down the filter's personal list. The filter does this without letting the commodity know what happened. The sender CxObject can be attached to a filter to notify a commodity that it received a CxMessage. CxSender() is a macro that creates a sender CxObject.

```
    senderCxObj = CxObj *CxSender(struct MsgPort *senderport, LONG cxmID);
```

CxSender() supplies the sender with an Exec message port and an ID. For every CxMessage a sender receives, it sends a new CxMessage to the Exec message port passed in CxSender(). Normally, the commodity creates this port. It is not unusual for a commodity's broker and sender(s) to share an Exec message port. The HotKey.c example does this to avoid creating unnecessary message ports. A sender uses the ID (cxmID) passed to CxSender() as the ID for all the CxMessages that the it transmits. A commodity uses the ID to monitor CxMessages from several senders at a single message port.

A sender does several things when it receives a CxMessage. First, it duplicates the CxMessage's corresponding input event and creates a new CxMessage. Then, it points the new CxMessage's data field to the copy of the input event and sets the new CxMessage's ID to the ID passed to CxSender(). Finally, it sends the new CxMessage to the port passed to CxSender(), asynchronously.

Because HotKey uses only one message port between its broker and sender object, it has to extract the CxMessage's type so it can tell if it is a CXM_IEVENT or a CXM_COMMAND. If HotKey gets a CXM_IEVENT, it compares the CxMessage's ID to the sender's ID, EVT_HOTKEY, to see which sender sent the CxMessage. Of course HotKey has only one sender, so it only checks for only one ID. If it had more senders, HotKey would check for the ID of each of the other senders as well.

Although HotKey doesn't use it, a CXM_IEVENT CxMessage contains a pointer to the copy of an input event. A commodity can extract this pointer (using CxMsgData()) if it needs to examine the input event copy. This pointer is only valid before the CxMessage reply. Note that it does not make any sense to modify the input event copy.

Senders are attached almost exclusively to CxObjects that filter out most input events (usually a filter CxObject). Because a sender sends a CxMessage for every single input event it gets, it should only get a select few input events. The AttachCxObj() function can add a CxObject to the end of a filter's (or some other filtering CxObject's) personal list. A commodity should not attach a CxObject to a sender as a sender ignores any CxObjects in its personal list.

1.12 31 Commodities Exchange Library / Translate CxObjects

Normally, after a commodity processes a hot key input event, it needs to eliminate that input event. Other commodities may need to replace an input event with a different one. The translate CxObject can be used for these purposes.

```
translateCxObj = CxObj  *CxTranslate(struct InputEvent *newinputevent);
```

The macro CxTranslate() creates a new translate CxObject. CxTranslate()'s only argument is a pointer to a chain of one or more InputEvent structures.

When a translate CxObject receives a CxMessage, it eliminates the CxMessage and its corresponding input event from the system. The translator introduces a new input event, which Commodities Exchange copies from the InputEvent structure passed to CxTranslate() (newinputevent from the function prototype above), in place of the deleted input event.

A translator is normally attached to some kind of filtering CxObject. If it wasn't, it would translate all input events into the same exact input event. Like the sender CxObject, a translator does not divert CxMessages down its personal list, so it doesn't serve any purpose to add any to it.

```
void SetTranslate( CxObj *translator, struct InputEvent *ie );
```

It is possible to change the InputEvent structure that a translator looks

at when it creates and introduces new input events into the input stream. The function SetTranslate() accepts a pointer to the new InputEvent structure, which the translator will duplicate and introduce when it receives a CxMessage.

HotKey utilizes a special kind of translator. Instead of supplying a new input event, HotKey passes a NULL to CxTranslate(). If a translator has a NULL new input event pointer, it does not introduce a new input event, but still eliminates any CxMessages and corresponding input events it receives.

1.13 31 Commodities Exchange Library / CxObject Errors

A Commodities Exchange function that acts on a CxObject records errors in the CxObject's accumulated error field. The function CxObjError() returns a CxObject's error field.

```
co_errorfield = LONG CxObjError( CxObj *co );
```

Each bit in the error field corresponds to a specific type of error. The following is a list of the currently defined CxObject errors and their corresponding bit mask constants.

Error Constant	Meaning
COERR_ISNULL	CxObjError() was passed a NULL.
COERR_NULLATTACH	Someone tried to attach a NULL CxObject to this CxObject.
COERR_BADFILTER	This filter CxObject currently has an invalid filter description.
COERR_BADTYPE	Someone tried to perform a type specific function on the wrong type of CxObject (for example calling SetFilter() on a sender CxObject).

The remaining bits are reserved by Commodore for future use. HotKey.c checks the error field of its filter CxObject to make sure the filter is valid. HotKey.c does not need to check the other objects with CxObjError() because it already makes sure that these other objects are not NULL, which is the only other kind of error the other objects can cause in this situation.

Commodities Exchange has a function that clears a CxObject's accumulated error field, ClearCxObjError().

```
void ClearCxObjError( CxObj *co );
```

A commodity should be careful about using this, especially on a filter. If a commodity clears a filter's error field and the COERR_BADFILTER bit is set, Commodities Exchange will think that the filter is OK and start sending messages through it.

1.14 31 Commodities Exchange Library / Uniqueness

When a commodity opens its broker, it can ask Commodities Exchange not to launch another broker with the same name (nb_Name). The purpose of the uniqueness feature is to prevent the user from starting duplicate commodities. If a commodity asks, Commodities Exchange will not only refuse to create a new, similarly named broker, but it will also notify the original commodity if someone tries to do so.

A commodity tells Commodities Exchange not to allow duplicates by setting certain bits in the nb_Unique field of the NewBroker structure it sends to CxBroker():

```
NBU_UNIQUE      bit 0
NBU_NOTIFY      bit 1
```

Setting the NBU_UNIQUE bit prevents duplicate commodities. Setting the NBU_NOTIFY bit tells Commodities Exchange to notify a commodity if an attempt was made to launch a duplicate. Such a commodity will receive a CXM_COMMAND CxMessage with an ID of CXCMD_UNIQUE when someone tries to duplicate it. Because the uniqueness feature uses the name a programmer gives a commodity to differentiate it from other commodities, it is possible for completely different commodities to share the same name, preventing the two from coexisting. For this reason, a commodity should not use a name that is likely to be in use by other commodities (like "filter" or "hotkey"). Instead, use a name that matches the commodity name.

When HotKey.c gets a CXCMD_UNIQUE CxMessage, it shuts itself down. HotKey.c and all the windowless commodities that come with the Release 2 Workbench shut themselves down when they get a CXCMD_UNIQUE CxMessage. Because the user will expect all windowless commodities to work this way, all windowless commodities should follow this standard.

When the user tries to launch a duplicate of a system commodity that has a window, the system commodity moves its window to the front of the display, as if the user had clicked the "Show" gadget in the controller program's window. A windowed commodity should mimic conventions set by existing windowed system commodities, and move its window to the front of the display.

1.15 31 Commodities Exchange Library / Signal CxObjects

A commodity can use a sender CxObject to find out if a CxMessage has "visited" a CxObject, but this method unnecessarily uses system resources. A commodity that is only interested in knowing if such a visitation took place does not need to see a corresponding input event or a CxMessage ID. Instead, Commodities Exchange has a CxObject that uses an Exec signal.

```
signalCxObj = CxObj  *CxSignal(struct Task *, LONG cx_signal);
```

CxSignal() sets up a signal CxObject. When a signal CxObject receives a CxMessage, it signals a task. The commodity is responsible for determining the proper task ID and allocating the signal. Normally, a

commodity wants to be signalled so it uses FindTask(NULL) to find it's own task address. Note that cx_signal from the above prototype is the signal number as returned by AllocSignal(), not the signal mask made from that number. For more information on signals, see the "Exec Signals" chapter.

The example Divert.c (shown a little later in this chapter) uses a signal CxObject.

1.16 31 Commodities Exchange Library / Custom CxObjects

Although the CxObjects mentioned so far take care of most of the input event handling a commodity needs to do, they cannot do it all. This is why Commodities Exchange has a custom CxObject. When a custom CxObject receives a CxMessage, it calls a function provided by the commodity.

```
customCxObj = CxObj  *CxCustom(LONG *customfunction(), LONG  cxmID);
```

A custom CxObject is the only means by which a commodity can directly modify input events as they pass through the Commodities network as CxMessages. For this reason, it is probably the most dangerous of the CxObjects to use.

```
A Warning About Custom CxObjects.
---------------------------------
Unlike the rest of the code a commodities programmer writes, the
code passed to a custom CxObject runs as part of the input.device
task, putting severe restrictions on the function.  No DOS or
Intuition functions can be called.  No assumptions can be made about
the values of registers upon entry.  Any function passed to
CxCustom() should be very quick and very simple, with a minimum of
stack usage.
```

Commodities Exchange calls a custom CxObject's function as follows:

```
void customfunction(CxMsg *cxm, CxObj *customcxobj);
```

where cxm is a pointer to a CxMessage corresponding to a real input event, and customcxobj is a pointer to the custom CxObject. The custom function can extract the pointer to the input event by calling CxMsgData(). Before passing the CxMessage to the custom function, Commodities Exchange sets the CxMessage's ID to the ID passed to CxCustom().

The following is an example of a custom CxObject function that swaps the function of the left and right mouse buttons.

```
custom = CxCustom(CxFunction, 0L)

/* The custom function for the custom CxObject.  Any code for a */
/* custom CxObj must be short and sweet. This code runs as part */
/* of the input.device task                                     */
#define CODEMASK (0x00FF & IECODE_LBUTTON & IECODE_RBUTTON)
void CxFunction(register CxMsg *cxm, CxObj *co)
{
    struct InputEvent *ie;
```

```
    UWORD mousequals = 0x0000;

    /* Get the struct InputEvent associated with this CxMsg.  Unlike
     * the InputEvent extracted from a CxSender's CxMsg, this is a
     *  *REAL* input event, be careful with it.
     */
    ie = (struct InputEvent *)CxMsgData(cxm);

    /* Check to see if this input event is a left or right mouse button
     * by itself (a mouse button can also be a qualifier).  If it is,
     * flip the low order bit to switch leftbutton <--> rightbutton.
     */
    if (ie->ie_Class == IECLASS_RAWMOUSE)
        if ((ie->ie_Code & CODEMASK) == CODEMASK)  ie->ie_Code ^= 0x0001;

    /* Check the qualifiers. If a mouse button was down when this */
    /* input event occurred, set the other mouse button bit.      */
    if (ie->ie_Qualifier & IEQUALIFIER_RBUTTON)  mousequals |=
                                                   IEQUALIFIER_LEFTBUTTON;
    if (ie->ie_Qualifier & IEQUALIFIER_LEFTBUTTON)  mousequals |=
                                                      IEQUALIFIER_RBUTTON;

    /* clear the RBUTTON and LEFTBUTTON qualifier bits */
    ie->ie_Qualifier &= ~(IEQUALIFIER_LEFTBUTTON | IEQUALIFIER_RBUTTON);

    /* set the mouse button qualifier bits to their new values */
    ie->ie_Qualifier |= mousequals;
}
```

1.17 31 Commodities Exchange Library / Debug CxObjects

The final CxObject is the debug CxObject. When a debug CxObject receives a CxMessage, it sends debugging information to the serial port using KPrintF().

```
    debugCxObj  = CxObj  *CxDebug(LONG ID);
```

The debug CxObject will KPrintF() the following information about itself, the CxMsg, and the corresponding InputEvent structure:

```
    DEBUG NODE: 7CB5AB0, ID: 2
     CxMsg: 7CA6EF2, type: 0, data 2007CA destination 6F1E07CB
    dump IE: 7CA6F1E
     Class 1
     Code 40
     Qualifier 8000
     EventAddress 40001802
```

There has to be a terminal connected to the Amiga's serial port to receive this information. See the KPrintF() Autodoc (debug.lib) for more details. Note that the debug CxObject did not work before V37.

1.18 31 Commodities Exchange Library / The IX Structure

Commodities Exchange does not use the input event description strings discussed earlier to match input events. Instead, Commodities Exchange converts these strings to its own internal format. These input expressions are available for commodities to use instead of the input description strings. The following is the IX structure as defined in <libraries/commodities.h>:

```
    #define IX_VERSION   2

    struct InputXpression {
       UBYTE   ix_Version;     /* must be set to IX_VERSION  */
       UBYTE   ix_Class;       /* class must match exactly   */
       UWORD   ix_Code;
       UWORD   ix_CodeMask;    /* normally used for UPCODE   */
       UWORD   ix_Qualifier;
       UWORD   ix_QualMask;
       UWORD   ix_QualSame;    /* synonyms in qualifier      */
       };
    typedef struct InputXpression IX;
```

The ix_Version field contains the current version number of the InputXpression structure. The current version is defined as IX_VERSION. The ix_Class field contains the IECLASS_ constant (defined in <devices/inputevent.h>) of the class of input event sought. Commodities Exchange uses the ix_Code and ix_CodeMask fields to match the ie_Code field of a struct InputEvent. The bits of ix_CodeMask indicate which bits are relevant in the ix_Code field when trying to match against a ie_Code. If any bits in ix_CodeMask are off, Commodities Exchange does not consider the corresponding bit in ie_Code when trying to match input events. This is used primarily to mask out the IECODE_UP_PREFIX bit of rawkey events, making it easier to match both up and down presses of a particular key.

IX's qualifier fields, ix_Qualifier, ix_QualMask, and ix_QualSame, are used to match the ie_Qualifier field of an InputEvent structure. The ix_Qualifier and ix_QualMask fields work just like ix_Code and ix_CodeMask. The bits of ix_QualMask indicate which bits are relevant when comparing ix_Qualifier to ie_Qualifier. The ix_QualSame field tells Commodities Exchange that certain qualifiers are equivalent:

```
    #define IXSYM_SHIFT  1  /* left- and right- shift are equivalent     */
    #define IXSYM_CAPS   2  /* either shift or caps lock are equivalent  */
    #define IXSYM_ALT    4  /* left- and right- alt are equivalent       */
```

For example, the input description string

```
    "rawkey -caps -lalt -relativemouse -upstroke ralt tab"
```

matches a tab upstroke or downstroke with the right Alt key pressed whether or not the left Alt, either Shift, or the Caps Lock keys are down. The following IX structure corresponds to that input description string:

```
    IX ix = {
        IX_VERSION,             /* The version */
        IECLASS_RAWKEY,         /* We're looking for a RAWKEY event */
```

```
        0x42,                   /* The key the usa0 keymap maps to a tab */
        0x00FF & (~IECODE_UP_PREFIX), /* We want up and down key presses */
        IEQUALIFIER_RALT,           /* The right alt key must be down */
        0xFFFF & ~(IEQUALIFIER_LALT | IEQUALIFIER_LSHIFT |
                   IEQUALIFIER_RSHIFT | IEQUALIFIER_CAPSLOCK |
                   IEQUALIFIER_RELATIVEMOUSE),
                   /* don't care about left alt, shift, capslock, or */
                   /* relativemouse qualifiers                       */
        IXSYM_CAPS  /* The shift keys and the capslock key */
                    /* qualifiers are all equivalent       */
    };
```

The CxFilter() macro only accepts a description string to describe an input event. A commodity can change this filter, however, with the SetFilter() and SetFilterIX() function calls.

```
    void SetFilter( CxObj *filter, UBYTE *descrstring );
    void SetFilterIX( CxObj *filter, IX *ix );
```

SetFilter() and SetFilterIX() change which input events a filter CxObject diverts. SetFilter() accepts a pointer to an input description string. SetFilterIX() accepts a pointer to an IX input expression. A commodity that uses either of these functions should check the filter's error code with CxObjError() to make sure the change worked.

The function ParseIX() parses an input description string and translates it into an IX input expression.

```
    errorcode = LONG ParseIX( UBYTE *descrstring, IX *ix );
```

Commodities Exchange uses ParseIX() to convert the description string in CxFilter() to an IX input expression. As was mentioned previously, as of commodities.library version 37.3, ParseIX() does not work with certain kinds of input strings.

1.19 31 Commodities Exchange Library / Controlling CxMessages

A Custom CxObject has the power to directly manipulate the CxMessages that travel around the Commodities network. One way is to directly change values in the corresponding input event. Another way is to redirect (or dispose of) the CxMessages.

```
    void DivertCxMsg ( CxMsg *cxm, CxObj *headobj, CxObj *retobj );
    void RouteCxMsg  ( CxMsg *cxm, CxObj *co );
    void DisposeCxMsg( CxMsg *cxm );
```

DivertCxMsg() and RouteCxMsg() dictate where the CxMessage will go next. Conceptually, DivertCxMsg() is analogous to a subroutine in a program; the CxMessage will travel down the personal list of a CxObject (headobj in the prototype) until it gets to the end of that list. It then returns and visits the CxObject that follows the return CxObject (the return CxObject in the prototype above is retobj). RouteCxMsg() is analogous to a goto in a program; it has no CxObject to return to.

DisposeCxMsg() removes a CxMessage from the network and releases its

resources. The translate CxObject uses this function to remove a CxMessage.

The example Divert.c shows how to use DivertCxMsg() as well as a signal CxObject.

divert.c

1.20 31 Commodities Exchange Library / New Input Events

Commodities Exchange also has functions used to introduce new input events to the input stream.

```
struct InputEvent *InvertString( UBYTE *string, ULONG *keymap );
void               FreeIEvents( struct InputEvent *ie );
void               AddIEvents( struct InputEvent *ie );
```

InvertString() is an amiga.lib function that accepts an ASCII string and creates a linked list of input events that translate into the string using the supplied keymap (or the system default if the key map is NULL). The NULL terminated string may contain ANSI character codes, an input description enclosed in angle (<>) brackets, or one of the following backslash escape characters:

```
\r -- return
\t -- tab
\ -- backslash
```

For example:

```
abc<alt f1>\rhi there.
```

FreeIEvents() frees a list of input events allocated by InvertString(). AddIEvents() is a commodities.library function that adds a linked list of input events at the the top of the Commodities network. Each input event in the list is made into an individual CxMessage. Note that if passed a linked list of input events created by InvertString(), the order the events appear in the string will be reversed.

PopShell.c

1.21 31 Commodities Exchange Library / Function Reference

The following are brief descriptions of the Commodities Exchange functions covered in this chapter. All of these functions require Release 2 or a later version of the Amiga operating system. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details on each function call.

Table 31-2: Commodities Exchange Functions

Function	Description
CxBroker()	Creates a CxObject of type Broker.
CxFilter()	Creates a CxObject of type Filter.
CxSender()	Creates a CxObject of type Sender.
CxTranslate()	Creates a CxObject of type Translate.
CxSignal()	Creates a CxObject of type Signal.
CxCustom()	Creates a CxObject of type Custom.
CxDebug()	Creates a CxObject of type Debug.
DeleteCxObj()	Frees a single CxObject
DeleteCxObjAll()	Frees a group of connected CxObjects
ActivateCxObj()	Activates a newly created CxObject in the commodities network.
SetTranslate()	Sets up substitution of one input event for another by translate CxObjects.
CxMsgType()	Finds the type of a CxMessage.
CxMsgData()	Returns the CxMessage data.
CxMsgID()	Returns the CxMessage ID.
CxObjError()	Returns the CxObject's accumulated error field.
ClearCxObjError()	Clear the CxObject's accumulated error field.
ArgArrayInit()	Create a Tool Types array from argc and argv (Workbench or Shell).
ArgArrayDone()	Free the resources used by ArgArrayInit().
ArgString()	Return the string associated with a given Tool Type in the array.
ArgInt()	Return the integer associated with a given Tool Type in the array.
AttachCxObj()	Attaches a CxObject to the end of a given CxObject's list.
InsertCxObj()	Inserts a CxObject in a given position in a CxObject's list.
EnqueueCxObj()	Inserts a CxObject in a CxObject's list by priority.
SetCxObjPri()	Sets a CxObject's priority for EnqueueCxObj().
RemoveCxObj()	Removes a CxObject from a list.
SetFilter()	Set a filter for a CxObject from an input description string.
SetFilterIX()	Set a filter for a CxObject from an IX data structure.
ParseIX()	Convert an input description string to an IX data structure.
DivertCxMsg()	Divert a CxMessage to one CxObject and return it to another.
RouteCxMsg()	Redirect a CxMessage to a new CxObject.
DisposeCxMsg()	Cancel a CxMessage removing it from the Commodities network.
InvertString()	Creates a linked list of input events that

```
|                     correspond to a given string.                    |
|      FreeIEvents()  Frees the linked list of input events created    |
|                     with InvertString().                             |
|       AddIEvents()  Converts a list of input events to CxMessages    |
|                     and puts them into the network.                  |
|______________________________________________________________________|
```