

# Introduction to Commodities Exchange

by John Orr

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.

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.

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. The custom input handler needs to take its next evolutionary step.

Commodities Exchange is that step. It provides a simple, standardized way to program and control custom input handlers. It is divided into three parts: an Exec library, a controller program, and a scanned library.

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. 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.

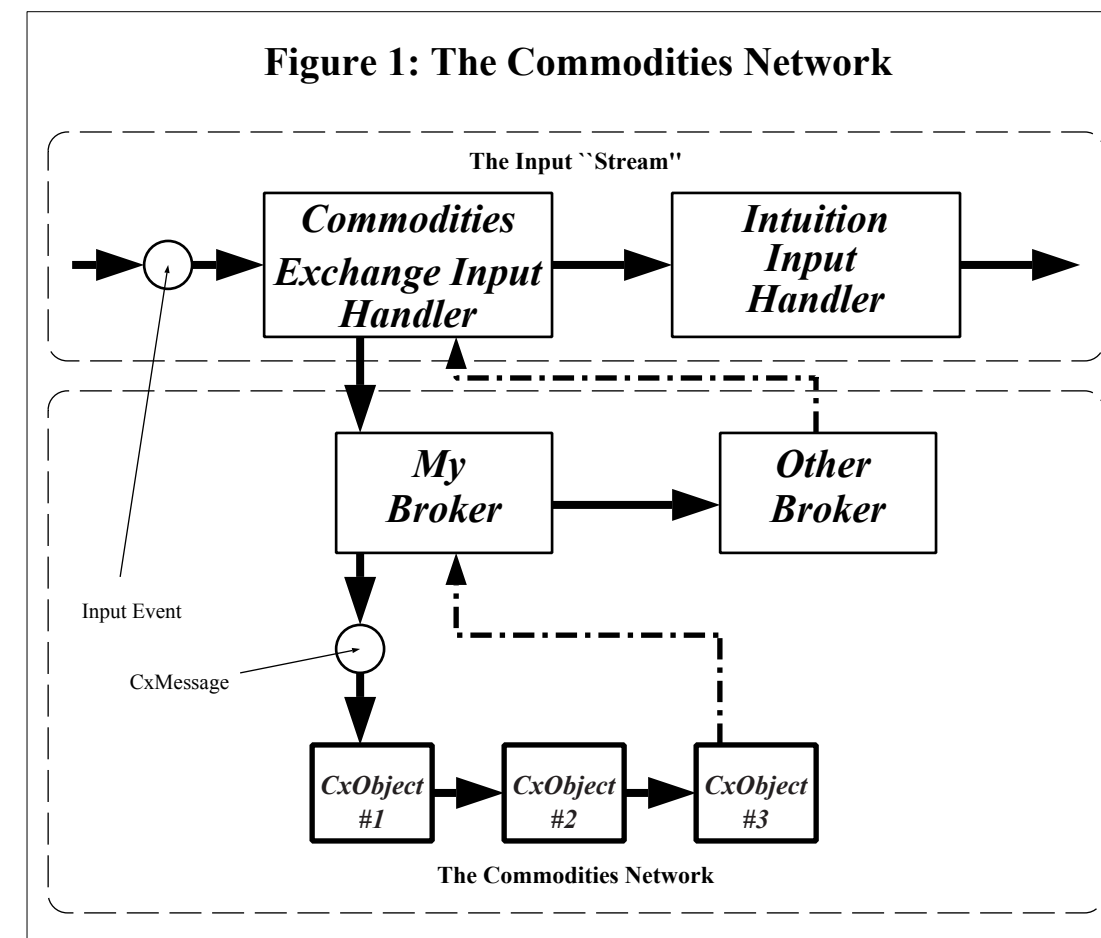
The third component of Commodities Exchange is its scanned library functions. These functions are part of the *amiga.lib* scanned library. They do a lot of the work involved with parsing command lines and tool types.

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 not 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.

## 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.



## Installing a Broker

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 a *broker*. 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.

Program listing 1, *Broker.c*, (located at the end of this article) 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;
};
```

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 article.

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 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 */
```

## 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 is an InputEvent structure. By using the CxMsgData() function, a commodity can obtain a pointer to the corresponding InputEvent of a CXM\_IEVENT. Commodities Exchange does not give an ID to the CXM\_IEVENT CxMessages it introduces to the Commodities network, but certain CxObjects can assign an ID to a CXM\_IEVENT CxMessage.

CXM\_COMMAND CxMessages generally do not use the data pointer. Instead, they use an ID. They use this ID to specify the command passed to a commodity. CxMsgID() extracts the ID from a CxMessage.

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

After successfully completing the initial set up and activating the broker using ActivateCxObj(), a commodity can begin its input processing loop. The example *Broker.c* receives input from one source, the controller program. The controller program sends a CxMessage each time the user clicks its enable, disable, or kill gadgets. Using the CxMsgID() function, the commodity finds out what the command is and executes it.

Notice that *Broker.c* uses Ctrl-C as a break key. This is a change from *1990 Atlanta DevCon Notes* on Commodities Exchange which said to use Ctrl-E. The break key for any commodity should be Ctrl-C.

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.

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

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

## Tool Types

A goal of Commodities Exchange is to improve user control over input handlers. One way in which it accomplishes this is through the use of standard ToolTypes. 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 ``popping up'' the commodity's window. The description string for CX\_POPKEY describes an input event. The specific format of the string is discussed later in this article.

Commodities Exchange's support library functions simplify parsing arguments from either the Workbench or a CLI. A Workbench launched commodity gets its arguments directly from the tool types in the commodity's icon. CLI 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 = char **ArgArrayInit(int argc, char **argv);
void ArgArrayDone(void);
tooltypevalue = char *ArgString(char **tooltypearray, char *tooltype, char *defaultvalue);
tooltypevalue = LONG *ArgInt(char **tooltypearray, char *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 CLI, 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.

### Filter CxObjects

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(char *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 a description string (CX\_POPKEY uses the same description string format):

```
[class] ( [-] (qual | syn) ) * [ [-] upstroke] [highmap |ANSIcode]
                                the * means zero or more occurances of ( [-] ( qual | syn ) )
```

where:

**class** can be any one of the strings in the table below. Each string corresponds to a class of input event (shown below). The classes are defined in *<devices/inpotevent.h>*. Commodities Exchange will assume the class is rawkey if the class is not explicitly stated.

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

**qual** is one of the strings in the table below. Each string corresponds to an input event qualifier (followed by its ID from *<devices/inpotevent.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 **qual** (or none at all) in the input description string.

"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
```

**syn** is one of the strings in the table below. These strings act as synonyms for groups of qualifiers. Each string below is followed by its identifier from *<libraries/commodities.h>*. A dash preceding the synonym 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 **syn** (or none at all) in the input description string.

```
"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 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.

The following are some example description strings:

```
``rawkey lshift alt f2"  Function key f2 with the left shift and either alt key pressed.
``-shift -alt -control a"    The key that produces an 'a' with or without any shift, alt, or
                             control keys pressed.
``rawmouse rbutton"        A mouse move with the right mouse button down.
``timer"                   A timer event.
```

As of *commodities.library* version 37.3, ParseIX(), the function used to parse input description strings, does not parse certain classes correctly. Only the rawkey, diskremoved, and newprefs classes are properly parsed. For the moment, any commodity that needs to

look for the other classes must directly use Commodities Exchange's internal format for input event descriptions, *input expressions* or *IX*, discussed later in this article.

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

```
void AttachCxObj(CxObj *HeadObj, CxObj *co);
```

This function uses the Exec function AddTail() to add the CxObject to the end of HeadObj's personal list. The position of a CxObject list determines which CxObject gets CxMessages first. Other functions exist to add CxObjects at different points in the list (see the *commodities.doc* Autodoc for details).

## Senders 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 is used 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.

## Translate CxObjects

Normally, after a commodity processes a hot key input event, it needs to eliminate the 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 `CxTranlate()` (`newinputevent` from the function prototype above), in place of the deleted input event. The translator introduces the new input event after the Commodities Exchange input handler in the *input.device's* input

stream. Any CxObjects that follow the translator in the Commodities network will see neither the deleted CxMessage nor the new 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.

*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.

## 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.

<code>COERR_ISNULL</code>	CxObjError() was passed a <code>NULL</code> .
<code>COERR_NULLATTACH</code>	Someone tried to attach a <code>NULL</code> CxObj to this CxObj.
<code>COERR_BADFILTER</code>	This CxObj is a filter and currently has an invalid filter description.
<code>COERR_BADTYPE</code>	Someone tried to perform a type specific function on the wrong kind of CxObject (for example calling <code>SetFilter()</code> on a sender CxObject).

The remaining bits are reserved by Commodore for future use. *HotKey* checks the error field of its filter CxObject to make sure the filter is valid. *HotKey* 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.

## Uniqueness

When a commodity opens its broker, it can ask Commodities Exchange not to launch another broker with the same name (`nb_Name`). This *uniqueness* feature's purpose 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`").

When *HotKey* gets a `CXCMD_UNIQUE` `CxMessage`, it shuts itself down. *HotKey* and all the windowless commodities that come with the 2.0 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.

## DeleteCxObjAll()

Cleaning up after a commodity requires deleting its `CxObjects`. 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.

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

*HotKey* 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.

## Other CxObjects

There are three remaining `CxObjects`:

```
signalCxObj = CxObj  *CxSignal(struct Task *,LONG signal);
debugCxObj  = CxObj  *CxDebug(LONG ID);
customCxObj = CxObj  *CxCustom(LONG *customfunction(), LONG cxmID);
```

When a signal `CxObject` receives a `CxMessage`, it signals a task. When a debug `CxObject` receives a `CxMessage`, it sends debugging information to the serial port using `kprintf()`. Note that the debug `CxObj` did not work before V37.

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`. The custom `CxObject` is probably the most dangerous of the `CxObjects` to use. 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()`.

## IX

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 (as defined in `<devices/inpotevent.h>`) of the particular 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_CodeMask` indicate which bits are relevant when comparing `ix_Qualifier` to `ie_Qualifier`. The `ix_QualSame` field tells

Commodities Exchange that certain qualifiers are equivalent. The bits of this field can be set to any of the following values:

```
#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 capslock 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.

```
void SetFilter( CxObj *filter, char *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.

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

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

This article is by no means an exhaustive description of Commodities Exchange. For more