

**commodities**

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> commodities	
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>
WRITTEN BY		July 18, 2024
<i>SIGNATURE</i>		

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>commodities</b>	<b>1</b>
1.1	commodities.doc	1
1.2	commodities.library/ActivateCxObj	1
1.3	commodities.library/AddIEvents	2
1.4	commodities.library/AttachCxObj	2
1.5	commodities.library/ClearCxObjError	3
1.6	commodities.library/CreateCxObj	3
1.7	commodities.library/CxBroker	4
1.8	commodities.library/CxMsgData	6
1.9	commodities.library/CxMsgID	6
1.10	commodities.library/CxMsgType	7
1.11	commodities.library/CxObjError	7
1.12	commodities.library/CxObjType	8
1.13	commodities.library/DeleteCxObj	9
1.14	commodities.library/DeleteCxObjAll	9
1.15	commodities.library/DisposeCxMsg	10
1.16	commodities.library/DivertCxMsg	10
1.17	commodities.library/EnqueueCxObj	11
1.18	commodities.library/InsertCxObj	11
1.19	commodities.library/InvertKeyMap	12
1.20	commodities.library/ParseIX	13
1.21	commodities.library/RemoveCxObj	13
1.22	commodities.library/RouteCxMsg	14
1.23	commodities.library/SetCxObjPri	14
1.24	commodities.library/SetFilter	15
1.25	commodities.library/SetFilterIX	15
1.26	commodities.library/SetTranslate	16

---

## Chapter 1

# commodities

### 1.1 commodities.doc

ActivateCxObj()	CxObjError()	ParseIX()
AddIEvents()	CxObjType()	RemoveCxObj()
AttachCxObj()	DeleteCxObj()	RouteCxMsg()
ClearCxObjError()	DeleteCxObjAll()	SetCxObjPri()
CreateCxObj()	DisposeCxMsg()	SetFilter()
CxBroker()	DivertCxMsg()	SetFilterIX()
CxMsgData()	EnqueueCxObj()	SetTranslate()
CxMsgID()	InsertCxObj()	
CxMsgType()	InvertKeyMap()	

### 1.2 commodities.library/ActivateCxObj

#### NAME

ActivateCxObj -- change the activation state of a commodity object.  
(V36)

#### SYNOPSIS

```
previous = ActivateCxObj(co,true);
          A0 D0
```

```
LONG ActivateCxObj(CxObj *,LONG);
```

#### FUNCTION

Commodity objects of all types maintain an "activation state". If an object is "active", then it performs its particular action whenever a commodity message arrives. If the object is "inactive" no action is taken, and the message goes on to its next destination.

All objects are created in the "active" state except brokers which are created "inactive". Thus, after you create your broker and hang a tree of objects off of it, you must remember to use this function to activate it. This causes it to divert all messages to your tree of objects.

This function activates 'co' if 'true' is different than 0. and

deactivates it otherwise. The previous activation state is returned.

#### INPUTS

co - the commodity object to affect  
true - 0 to deactivate the object, anything else to activate it

#### RESULTS

previous - the previous activation state: 0 if the object was inactive, anything else if the object was active

#### SEE ALSO

CxBroker()

### 1.3 commodities.library/AddIEvents

#### NAME

AddIEvents -- add input events to commodities' input stream. (V36)

#### SYNOPSIS

```
AddIEvents(events)
           A0
```

```
VOID AddIEvents(struct InputEvent *);
```

#### FUNCTION

This function adds a null-terminated linked list of input events to the input stream of commodities. It is a touch easier than using the input device directly.

The contents of the input events are copied into commodity messages, so they may be disposed of as soon as this call returns.

The messages are initially routed to the first broker in the Commodities Object List.

#### INPUTS

events - the list of input events to add

#### SEE ALSO

cx\_lib/FreeIEvents()

### 1.4 commodities.library/AttachCxObj

#### NAME

AttachCxObj -- attach a commodity object to the end of an existing list of objects. (V36)

#### SYNOPSIS

```
AttachCxObj(headObj,co);
           A0      A1
```

```
VOID AttachCxObj(CxObj *, CxObj *);
```

**FUNCTION**

Adds 'co' to the list of objects pointed to by 'headObj'. The new object is added at the end of the list.

**INPUTS**

headObj - pointer to a list of objects  
co - the object to add to the list

**RESULTS**

If 'co' is NULL, this function will record that fact in the internal accumulated error of 'headObj'. This error record can be retrieved using CxObjError() and cleared using ClearCxObjError().

**SEE ALSO**

CxObjError(), ClearCxObjError()

## 1.5 commodities.library/ClearCxObjError

**NAME**

ClearCxObjError -- clear the accumulated error value of a commodity object. (V36)

**SYNOPSIS**

```
ClearCxObjError(co);  
A0
```

```
VOID ClearCxObjError(CxObj *);
```

**FUNCTION**

This function clears the accumulated error value of commodity object 'co'.

It is unwise to do this to a filter if COERR\_BADFILTER is set. This will fool commodities into thinking the filter is OK. Set another valid filter, or leave the error value alone.

**INPUTS**

co - the object to affect

**SEE ALSO**

CxObjError()

## 1.6 commodities.library/CreateCxObj

**NAME**

CreateCxObj -- create a new commodity object. (V36)

**SYNOPSIS**

```
co = CreateCxObj(type, arg1, arg2);
```

---

D0 A0 A1

```
CxObj *CreateCxObj(ULONG, LONG, LONG);
```

#### FUNCTION

This function creates a commodity object of type 'type'. It is not proper to call this function directly. Each object creation routine except CxBroker() is defined as a macro in <libraries/commodities.h> These are independently documented.

All functions which operate on a commodity object are made with a reference to the thirty-two bit value returned by this function (or by CxBroker()).

#### INPUTS

type - the type of object to create as defined in <libraries/commodities.h>  
 arg1 - first argument, meaning depends on 'type'  
 arg2 - second argument, meaning depends on 'type'

#### RESULTS

co - a pointer to the new object or NULL if it could not be created. A NULL return indicates typically a lack of system memory. Minor problems in creating an object, such as providing a bad filter description to cx\_lib/CxFilter(), typically don't cause failure, but are recorded in an internal error field in the new object which can be accessed via CxObjError().

#### SEE ALSO

CxObjError(), cx\_lib/CxFilter(), cx\_lib/CxSender(), cx\_lib/CxSignal(), cx\_lib/CxTranslate(), cx\_lib/CxDebug(), cx\_lib/CxCustom(), CxBroker()

## 1.7 commodities.library/CxBroker

#### NAME

CxBroker -- create a commodity broker. (V36)

#### SYNOPSIS

```
broker = CxBroker(nb,error);
        A0 D0
```

```
CxObj *CxBroker(struct NewBroker *,LONG *);
```

#### FUNCTION

This function creates a broker from the specification found in the NewBroker structure pointed to by 'nb'. The purpose and meaning of the fields of the NewBroker structure are described below. Also see <libraries/commodities.h> for more info.

```
struct NewBroker
{
    BYTE          nb_Version;
    BYTE          *nb_Name;
    BYTE          *nb_Title;
```

```
    BYTE          *nb_Descr;
    SHORT         nb_Unique;
    BYTE          nb_Pri;
    struct MsgPort *nb_Port;
    WORD          nb_ReservedChannel;
};
```

#### nb\_Version

This is the way that future versions of `commodities.library` can identify which version of the `NewBroker` structure you are using. This should be set to `NB_VERSION`, defined in `<libraries/commodities.h>`

#### nb\_Name

This is the name of the broker. This name is used to find the broker in the Commodities Object List and is the name shown in the list gadget of the Exchange program. The name string is copied in the broker object upon creation, so this string can be discarded right after `CxBroker()` returns. The maximum length of the name string is defined by a constant in `<libraries/commodities.h>`.

#### nb\_Title, nb\_Descr

These are two strings which appear to the user in the Exchange program and describe the application the broker is representing. Note that these strings are copied into the broker object. The maximum length of these strings that will be recognized are defined by constants in `<libraries/commodities.h>`.

#### nb\_Unique

This field indicates what should happen if a broker of the same name (`nb_Name`) already exists in the Commodities Object List. Constants in `<libraries/commodities.h>` allow the caller to specify whether another broker is to be created, and whether any existing broker of the same name should be notified that an attempt at creating a duplicate has been made.

#### nb\_Pri

This specifies with what priority the new broker is to be inserted within the Commodities Object List. Higher priority nodes appear earlier in a list. See `<libraries/commodities.h>` for guidelines for priorities of different types of applications. It is strongly recommended that the `ToolTypes` environment of an application be used to allow the end-user to set the priority of the broker.

#### INPUTS

`nb` - an initialized `NewBroker` structure  
`error` - a pointer to a longword where to store a failure code, this may be `NULL`

#### RESULTS

`broker` - a pointer to the broker object or `NULL` upon failure. If the 'error' is not `NULL`, a further diagnostic code is placed at that address. Error codes are in `<libraries/commodities.h>` and include:

`CBERR_OK`

No problems; broker created OK.

---

CBERR\_SYSERR  
System problems, not your fault. Sign of low memory or big problems.

CBERR\_DUP  
The nb\_Unique field specified that only one broker of 'nb\_Name' should be allowed, and one already exists.

CBERR\_VERSION  
The version specified in 'nb\_Version' is unknown to the library.

SEE ALSO  
SetCxObjPri(), <libraries/commodities.h>

## 1.8 commodities.library/CxMsgData

### NAME

CxMsgData -- obtain a pointer to a commodity message's data area. (V36)

### SYNOPSIS

```
data = CxMsgData(cxm);
      A0
```

```
UBYTE *CxMsgData(struct CxMsg *);
```

### FUNCTION

Most commodity messages contain meaningful data, such as an InputEvent structure. This function returns a pointer to this data.

You may get a commodity message from a synchronous (custom object) or asynchronous (sender object) source. In the second case, 'data' is not valid after you have replied to the message.

### INPUTS

cxm - the commodity message to get the data pointer from

### RESULTS

data - a pointer to the message's data, or NULL if 'cxm' is NULL.

### SEE ALSO

cx\_lib/CxSender(), cx\_lib/CxCustom()

## 1.9 commodities.library/CxMsgID

### NAME

CxMsgID -- obtain the ID of a commodity message. (V36)

### SYNOPSIS

```
id = CxMsgID(cxm);
    A0
```

```
LONG CxMsgID(struct CxMsg *);
```

#### FUNCTION

This function returns the value associated with the cause or source of the commodity message 'cxm'. Values are provided by the application when a sender or custom object is created.

#### INPUTS

cxm - the commodity message to inquire about

#### RESULTS

id - if not specified by the application, the ID value of a commodity message will be 0. It is suggested that using non-zero values in your program as a rule may identify some possible errors.

#### SEE ALSO

cx\_lib/CxSender(), cx\_lib/CxCustom()

## 1.10 commodities.library/CxMsgType

#### NAME

CxMsgType -- obtain the type of a commodity message. (V36)

#### SYNOPSIS

```
type = CxMsgType(cxm);  
A0
```

```
ULONG CxMsgType(struct CxMsg *);
```

#### FUNCTION

This function returns the type of a commodity message. Possible values of 'type' are defined in <libraries/commodities.h>. Most commodity messages are of type CXM\_IEVENT.

#### INPUTS

cxm - the commodity message to inquire about

#### RESULTS

type - the type of the commodity message, possible values are defined in <libraries/commodities.h>

## 1.11 commodities.library/CxObjError

#### NAME

CxObjError -- obtain a commodity object's accumulated error. (V36)

#### SYNOPSIS

```
error = CxObjError(co);  
A0
```

```
LONG CxObjError(CxObj *);
```

---

**FUNCTION**

When a function acting on an object fails, it records the failure in the object. This function returns the accumulated error value. The values are represented by flag bits defined in `<libraries/commodities.h>`. Several errors may be recorded by multiple bits in 'error'.

The currently defined errors are:

**COERR\_ISNULL**

The value of parameter 'co' was in fact NULL. This error means "the problem with the object you inquire about is that it failed to be created."

**COERR\_NULLATTACH**

Using the commodities' list manipulation functions, an attempt was made to add a NULL object to the list belonging to 'co'. This allows a line of code as follows to exist in an error-tolerant program:

```
AttachCxObj(filter,CxSender(myport,MY_ID));
```

**COERR\_BADFILTER**

The most recent filter specification for a filter object was faulty. This happens if no sense can be made out of a description string, or if an input expression (IX) has an invalid format or version byte. When this bit is set in a filter's error field, the filter will match nothing, but this is not the proper way to "turn off" a filter, use `ActivateCxObj()`.

**COERR\_BADTYPE**

A type specific operation, such as `SetFilterIX()`, was called for object 'co', but 'co' isn't of the proper type.

**INPUTS**

co - the commodity object to get the accumulated error from

**RESULTS**

error - the accumulated error

**SEE ALSO**

`SetFilter()`, `SetFilterIX()`, `AttachCxObj()`, `ActivateCxObj()`, `ClearCxObjError()`

## 1.12 commodities.library/CxObjType

**NAME**

CxObjType -- obtain the type of a commodity object. (V36)

**SYNOPSIS**

```
type = CxObjType(co);
      A0
```

```
ULONG CxObjType(CxObj *);
```

---

**FUNCTION**

This function should not really be necessary. It returns the type of a commodity object, which you should already know, since you created it in the first place.

**INPUTS**

co - the commodity object to inquire about

**RESULTS**

type - the type of the commodity object, possible values are defined in <libraries/commodities.h>

**SEE ALSO**

CreateCxObj()

## 1.13 commodities.library/DeleteCxObj

**NAME**

DeleteCxObj -- delete a commodity object. (V36)

**SYNOPSIS**

```
DeleteCxObj(co);  
          A0
```

```
VOID DeleteCxObj(CxObj *);
```

**FUNCTION**

Deletes a commodity object of any type. If the object is linked into a list, it is first removed. Note that the handle 'co' is invalid after this function is called.

Also note that deleting an object which has other objects attached to it may be undesirable. Use the function DeleteCxObjAll() to delete an entire sub-tree of objects.

**INPUTS**

co - the commodity object to delete

**SEE ALSO**

DeleteCxObjAll()

## 1.14 commodities.library/DeleteCxObjAll

**NAME**

DeleteCxObjAll -- recursively delete a tree of commodity objects. (V36)

**SYNOPSIS**

```
DeleteCxObjAll(co);  
          A0
```

```
VOID DeleteCxObjAll(CxObj *);
```

---

## FUNCTION

This function deletes the commodity object 'co', and recursively deletes all objects attached to it, and the objects attached to them, etc.

If 'co' is linked into a list, it is first removed. Note that the handle 'co' is invalid after this function is called.

This function is useful when an application exits: most applications can clean up completely by deleting the entire sub-tree of objects starting at their broker.

## INPUTS

co - the first commodity object to delete

## SEE ALSO

DeleteCxObj()

## 1.15 commodities.library/DisposeCxMsg

## NAME

DisposeCxMsg -- delete a commodity message. (V36)

## SYNOPSIS

```
DisposeCxMsg(cxm);
           A0
```

```
VOID DisposeCxMsg(struct CxMsg *);
```

## FUNCTION

This function eliminates the commodity message pointed to by 'cxm'. Can be used to 'swallow' InputEvents by disposing of every commodity message of type CXM\_IEVENT.

## INPUTS

cxm - the commodity message to delete

## 1.16 commodities.library/DivertCxMsg

## NAME

DivertCxMsg -- send a commodity message down an object list. (V36)

## SYNOPSIS

```
DivertCxMsg(cxm, headObj, returnObj);
           A0  A1      A2
```

```
VOID DivertCxMsg(struct CxMsg *, CxObj *, CxObj *);
```

## FUNCTION

This function sends the commodity message 'cxm' down the list of objects attached to 'headObj'. The pointer 'returnObj' is first

pushed onto the routing stack of 'cxm' so that when the end of the list of 'headObj' is reached the SUCCESSOR of 'returnObj' is the next destination.

For example, when a filter finds a match with a message, the message is diverted down the filter's list like this:

```
DivertCxMsg(cxm,filter,filter);
```

#### INPUTS

cxm - the commodity message to divert  
headObj - the list of objects to divert the message to  
returnObj - the object to use as a place holder

#### SEE ALSO

RouteCxMsg()

## 1.17 commodities.library/EnqueueCxObj

#### NAME

EnqueueCxObj -- insert a commodity object within a list of objects based on object priority. (V36)

#### SYNOPSIS

```
EnqueueCxObj(headObj,co);  
             A0      A1
```

```
VOID EnqueueCxObj(CxObj *,CxObj *);
```

#### FUNCTION

This function puts object 'co' into the list of object 'headObj'. The insertion point is determined by the object's priority. The objects are kept in the list from the highest priority to the lowest. New nodes are inserted in front of the first node with a lower priority. Hence a FIFO queue for nodes of equal priority.

The priority of the commodity object can be set using SetCxObjPri().

#### INPUTS

headObj - pointer to a list of objects  
co - the object to add to the list

#### RESULTS

If 'co' is NULL, this function will record that fact in the internal accumulated error of 'headObj'. This error record can be retrieved using CxObjError() and cleared using ClearCxObjError().

#### SEE ALSO

SetCxObjPri(), CxObjError(), ClearCxObjError()

## 1.18 commodities.library/InsertCxObj

---

## NAME

InsertCxObj -- insert a commodity object in a list after a given object. (V36)

## SYNOPSIS

```
InsertCxObj(headObj,co,pred);
           A0      A1 A2
```

```
VOID InsertCxObj(CxObj *,CxObj *,CxObj *);
```

## FUNCTION

Adds 'co' to the list of objects pointed to by 'headObj' after object 'pred'.

## INPUTS

headObj - pointer to a list of objects  
 co - the object to insert in the list  
 pred - the object after which 'co' should be inserted. If this is NULL then 'co' is added to the head of the list.

## RESULTS

If 'co' is NULL, this function will record that fact in the internal accumulated error of 'headObj'. This error record can be retrieved using CxObjError() and cleared using ClearCxObjError().

## SEE ALSO

CxObjError(), ClearCxObjError()

## 1.19 commodities.library/InvertKeyMap

## NAME

InvertKeyMap -- generate an input event from an ANSI code. (V36)

## SYNOPSIS

```
success = InvertKeyMap(ansicode,event,km)
           D0      A0      A1
```

```
BOOL InvertKeyMap(ULONG,struct InputEvent *,struct KeyMap *);
```

## FUNCTION

This function uses the system call MapANSI() to figure out what InputEvent translates to an ANSI character code 'ansicode'. The InputEvent pointed to by 'event' is filled in with that information. The KeyMap 'km' is used for the translation, unless it is NULL, in which case the system default keymap (as defined when commodities.library is initialized) is used.

## INPUTS

ansicode - the ANSI code to convert to an input event  
 event - the InputEvent to fill-in  
 km - the keymap to use for the translation, or NULL to use the system default keymap.

## RESULTS

success - TRUE if the translation worked, FALSE otherwise.

## BUGS

This function currently handles one-deep dead keys (such as <alt f>). It does not look up the high key map (keystrokes with scan codes greater than 0x40), and misses system changes to the default key map.

## SEE ALSO

cx\_lib/InvertString()

## 1.20 commodities.library/ParseIX

## NAME

ParseIX -- initialize an input expression given a description string.  
(V36)

## SYNOPSIS

```
failureCode = ParseIX(description,ix);
                    A0          A1
```

```
LONG ParseIX(STRPTR,IX *);
```

## FUNCTION

Given an input description string and an allocated input expression, sets the fields of the input expression to correspond to the description string.

## INPUTS

description - the string to parse  
ix - the input expression to hold the result of the parse

## RESULTS

failureCode - 0 if all went well,  
              -1 if tokens after end (code spec)  
              -2 if 'description' was NULL

## 1.21 commodities.library/RemoveCxObj

## NAME

RemoveCxObj -- remove a commodity object from a list. (V36)

## SYNOPSIS

```
RemoveCxObj(co);
            A0
```

```
VOID RemoveCxObj(CxObj *);
```

## FUNCTION

This function remove 'co' from any list it may be a part of.  
Will not crash if 'co' is NULL, or if it has not been inserted

---

in a list (and is not corrupted).

It is not recommended to remove a broker from the master list.

#### INPUTS

co - the object to remove, may be NULL

#### SEE ALSO

AttachCxObj(), EnqueueCxObj(), InsertCxObj()

## 1.22 commodities.library/RouteCxMsg

#### NAME

RouteCxMsg -- set the next destination of a commodity message. (V36)

#### SYNOPSIS

```
RouteCxMsg(cxm,co);
           A0 A1
```

```
VOID RouteCxMsg(struct CxMsg *,CxObj *);
```

#### FUNCTION

Establishes the next destination of a commodity message to be 'co', which must be a valid commodity object, and must be linked in ultimately to the Commodities Object List.

Routing of an object is analogous to a 'goto' in a program. There is no effect on the message's routing stack.

#### INPUTS

cxm - the commodity message to route  
co - the commodity object to route the message to

#### SEE ALSO

DivertCxMsg()

## 1.23 commodities.library/SetCxObjPri

#### NAME

SetCxObjPri -- set the priority of a commodity object. (V36)

#### SYNOPSIS

```
SetCxObjPri(co,pri)
           A0 D0
```

```
VOID SetCxObjPri(CxObj *,LONG);
```

#### FUNCTION

This function sets the priority of a commodity object for the purposes of EnqueueCxObj().

It is strongly recommended that the ToolTypes environment be

---

utilized to provide end-user control over the priority of brokers, but application specific ordering of other objects within their lists is not dictated.

**INPUTS**

co - the commodity object to affect  
pri - the object's new priority in the range -128 to +127. A value of 0 is normal.

**SEE ALSO**

EnqueueCxObj()

## 1.24 commodities.library/SetFilter

**NAME**

SetFilter -- change the matching condition of a commodity filter.  
(V36)

**SYNOPSIS**

```
SetFilter(filter, text);
           A0      A1
```

```
VOID SetFilter(CxObj *, STRPTR);
```

**FUNCTION**

This function changes the matching condition of a commodity input filter to that described by the input description string 'text'.

**INPUTS**

filter - the filter object to affect  
text - the new matching conditions for the filter

**RESULTS**

The internal error of 'filter' will have the COERR\_BADFILTER bit set or cleared depending on the failure or success of this function.

**SEE ALSO**

SetFilterIX(), CxObjError()

## 1.25 commodities.library/SetFilterIX

**NAME**

SetFilterIX -- change the matching condition of a commodity filter.  
(V36)

**SYNOPSIS**

```
SetFilterIX(filter, ix);
           A0      A1
```

```
VOID SetFilterIX(CxObj *, IX *);
```

---

## FUNCTION

This function changes the matching condition of a commodity input filter to that described by the binary input expression pointed by 'ix'.

Input expressions are defined in <libraries/commodities.h>. It is important to remember that the first field of the input expression structure must indicate which version of the input expression structure is being used.

## INPUTS

filter - the filter object to affect  
ix - the new matching conditions for the filter

## RESULTS

The internal error of 'filter' will have the COERR\_BADFILTER bit set or cleared depending on the failure or success of this function.

## SEE ALSO

SetFilter(), CxObjError(), <libraries/commodities.h>

## 1.26 commodities.library/SetTranslate

## NAME

SetTranslate -- replace a translator object's translation list. (V36)

## SYNOPSIS

```
SetTranslate(translator,events);  
           A0           A1
```

```
VOID SetTranslate(CxObj *,struct InputEvent *);
```

## FUNCTION

This function replaces the translation list of a commodity translator object with the linked list starting at 'events'.

A NULL value for 'events' indicates that the object 'translator' should swallow all commodity messages that are sent its way.

Note that the input events are not copied into commodities' private memory, but the value of 'events' is used -- asynchronously to the application program -- to find a chain of InputEvents in the application's data space. At the time of translation, each input event is copied into its own new commodity message.

The above means that no other commodities' user, nor commodities.library itself will be modifying your list of InputEvents. On the other hand, your program must not corrupt the input event chain that has been presented to a translator.

## INPUTS

translator - the translator object to affect  
events - the new input event translation list

---

SEE ALSO

`<devices/inputevent.h>`, `cx_lib/CxTranslate()`

---