X·IPC Message Oriented Middleware

Version 3

Scalable Message Oriented Middleware for Distributed Computing

Technical Notes

CONTENTS

THE *X♦IPC* IDLE USER DETECTION MECHANISM

TABLE OF CONTENTS

The X♦IPC Idle User Detection Mechanism

1.1 Background

Computer applications operating in the context of a client/server architecture typically require that the client and server components be kept abreast of each other's respective status. A primary motivation for this requirement is to allow the server program to recycle server resources whenever a participating client "goes away."
While this requirement is most evident within *networked* applications (i.e., where clients and servers are physically separated by a network), it is similarly a concern in situations where the processes are operating on a *single* platform. Nevertheless, the penalties for not addressing this requirement properly within a distributed environment are more severe than within a stand-alone environment and, if left unaddressed, such distributed applications can exhibit one or both of the following problems:

- ❑ Network sessions are kept open after the application's need for them has ended.
- ❑ Server operating system processes and other resources are held onto after they are no longer needed.

Distributed applications involving "not so intelligent" client platforms (e.g., MS/Windows), are particularly susceptible to this problem. Over time, as network and operating resources are consumed, and left idle, such a problem can bring a distributed application to its knees.

1.2 The X♦IPC Specific Problem

This problem can manifest itself when working with X♦IPC in that it is possible for X♦IPC user programs to hold network, operating system and X♦IPC instance resources following ungraceful user program termination.
An example of this within a network environment is when a user of an X♦IPC -based distributed application "powers down" an MS/Windows machine without first exiting the applications being run (i.e., without first logging out of the X♦IPC instances being used by the application). This *modus operandi* is more often the rule than the exception. The result, within an X♦IPC network environment, is that not all X♦IPC, system and network resources are released.

1.3 The X♦IPC Solution: The Idle User Detection Mechanism

The X♦IPC Idle User Detection Mechanism makes it possible for X♦IPC to automatically monitor and detect *idle users* within an instance and to cause the release of any resources held by such users. In order for X♦IPC users within an instance to be subject to such monitoring and to potential resource recovery, two prerequisites must be satisfied:

- ❑ The instance involved must be actively monitored against idle user activity, and
- ❑ Each instance user must explicitly request that it be monitored against idle activity within that instance.

If *either* of these conditions is not met, then users will not be subject to any form of idle usage monitoring. This provides maximum flexibility from both the user program and the instance perspective for controlling the operation of the idle user detection mechanism.

2. CONFIGURATION AND PROGRAMMING CONCEPTS

Operation of the idle-user detection mechanism is controlled by two components: the `xipcidld` daemon program and the `XipcIdleWatch()` function call.

2.1 The `xipcidld` Daemon Program

The `xipcidld` daemon program is used for monitoring user activity within an XIPC instance so as to detect the presence of idle users within that instance. Resources held by such users are forcibly recovered. A user is determined to be idle within an instance when it has not executed an XIPC operation against the instance, over a certain period of time. This time period is referred to as the *idle-interval* of that instance. Each instance is assigned its own *idle-interval* value.

2.1.1 STARTING `xipcidld`

The `xipcidld` daemon is started on the platform that serves the instance via the `xipcinit` command.

2.1.2 STOPPING `xipcidld`

The `xipcidld` daemon is stopped via the `xipcterm` command.

2.1.3 REGISTERING AN INSTANCE

The `[IDLE_USER]` section must be defined in the configuration file if the instance is to be monitored with `xipcidld` at `xipcstart`. If `[IDLE_USER]` is not defined, the instance will not be monitored.
The `[IDLE_USER]` section may specify values for any of the following parameters; if the parameter is omitted, its default value is used:

Parameter	Description	Default Value
`INTERVAL`	How often an instance should be monitored for idle users, such as `30m, 30s, 10h`.	`30m`
`LOGFILENAME`	The name of a file to log idle information for that instance.	No log
`ACTION`	The action the `xipcidld` should take when a user is found to be idle. The options are: `NOACTION`, `ABORT` and *`UserDefinedExitProgram`*.	`ABORT`

It is possible with the current version of XIPC to monitor multiple XIPC instances on a single platform. (Previously, only a single instance per platform could be specified in the `xipc.env` file for idle user monitoring; this option is still supported for backward compatibility.)

2.2 The `XipcIdleWatch()` Function

By default, users logging into an X•IPC instance that is monitoring against idle users, are *not* subject to the monitoring. Users wishing to have their activity within the instance monitored *must* notify that instance of such desire. This is accomplished via the `XipcIdleWatch()` function call, which is fully defined in the the X•IPC Reference Manual.

The `XipcIdleWatch()` function call can be used to toggle the user's state within an instance - between being watched and not being watched. It is additionally possible for a user to notify an instance that it is *still alive* even though it has not recently performed X•IPC operations within the instance. This too is accomplished using the `XipcIdleWatch()` function.

The `XipcIdleWatch()` function takes one argument, *`WatchOption`*, that can have one of three values:

- ❑ The `XIPC_IDLEWATCH_START` value notifies the instance to start monitoring the calling user as part of the instance's idle user detection activity.
- ❑ The `XIPC_IDLEWATCH_STOP` value notifies the instance to stop monitoring the calling user as part of the instance's idle user detection activity.
- ❑ The `XIPC_IDLEWATCH_MARK` argument value notifies the instance that the calling user is still alive. Calling `XipcIdleWatch()` with this option is a means of issuing a "heartbeat" to the instance. This saves the user from detection during the current idle-period cycle. Another way of viewing this option is as if the user is executing a null X•IPC operation within the instance.

All calls to `XipcIdleWatch()` are ignored if the instance is not currently watching for idle users.

2.2.1 LOG FILES

General `xipcidld` information and any general errors are logged to the `xipcidld.log` file. This file is located within the X•IPC platform log directory. (See the X•IPC User Guide, section 3.4.)

Specific log errors for an instance are logged to the user-specified log file, if specified. Note that if multiple instances are being monitored, each instance should have a different log file.

3. AN EXAMPLE OF USING THE *X•IPC* IDLE USER DETECTION MECHANISM

3.1.1 STARTING `xipcidld`

`xipcidld` is one of the default processes started by `xipcinit`. Refer to the X•IPC Reference Manual for a discussion of how to start the X•IPC platform without `xipcidld`. The `.cfg` file of the instance to be monitored must contain an `[IDLE_USER]` section. If the default parameter values are not to be used, then paramtere values for monitoring the instance must be specified in this section. If the default parameter values are to be used, the `[IDLE_USER]` section can be left empty.

3.1.2 SAMPLE USER PROGRAM

The following sample user program demonstrates the usage of the `XipcIdleWatch()` function for controlling the user's susceptibility to being monitored:

```
#include <xipc.h>

VOID
main()
{
    /*
     * Login to the "example"  instance. By default, the user will initially
     * not be susceptible to the instance's idle-user monitoring.
     */

    XipcLogin("@example", ... );
    ...
    ...
    /*
     * Start to be monitored.
     */

     XipcIdleWatch(XIPC_IDLEWATCH_START);

    ...
    ...

    /*
     * It's been some time since executing an XIPC operation against the
     * "example" instance. Let it know that I'm still alive.
     */

     XipcIdleWatch(XIPC_IDLEWATCH_MARK);

    ...

    /*
     * Stop being monitored.
     */

     XipcIdleWatch(XIPC_IDLEWATCH_STOP);

     XipcLogout ();
}
```

3.2 Summary of Idle User Enhancements in *X♦IPC* Version *3.0.1*

- ♦ The parameter `MAX_INSTANCES` can now be included in the `[xipcidld]` section of the `xipc.env` file. This specifies the maximum number of instances that may be simultaneously monitored for idle users. The default value is 10.
- ♦ The `xipcidld` is now started as one of the default daemon/service programs by `XIPCINIT`.
- ♦ It is now possible to monitor multiple *X♦IPC* instances on a single platform. Previously, only a single instance per platform could be specified for idle user monitoring; this option is still supported for backward compatibility. Each instance must have an `[IDLE_USER]` section in its `.cfg` file, indicating to the `xipcidld` daemon program that the instance is to be monitored for idle users.
- ♦ The `INSTANCENAME` parameter of the `[xipcidld]` section of the `xipc.env` file no longer has a default value. (It had been the value of the `XIPC` environment variable.) The preferred way to specify idle user monitoring for an instance is to by including an `[IDLE_USER]` section in the instance's `.cfg` file, rather than by specifying the instance name in the `xipc.env` file's `[xipcidld]` section.

USING I/O DESCRIPTORS FOR ASYNCHRONOUS OPERATIONS ON *UNIX*

TABLE OF CONTENTS

1. *X•IPC* ASYNCHRONOUS OPERATIONS

A major advantage of developing multitasking and distributed applications using *X•IPC* is that it provides a rich set of asynchronous functionality. The benefits of such mechanisms are many, the most significant being that they allow an application's distributed processes to execute *concurrently* on a single multitasking platform as well as on multiple network nodes, thus leveraging the inherent parallelism provided by such environments.

A key step in the asynchronous execution of *X•IPC* operations is that of *completion notification*. This is the step by which *X•IPC* notifies a process of the completion of its asynchronous *X•IPC* operations. On designated platforms, *X•IPC* currently supports two completion notification methods.

1.1 Using Signals

As its default method, *X•IPC* implements completion notification by means of the native operating system's process signaling and interrupt mechanism. *X•IPC* signals the involved process that an asynchronous *X•IPC* operation has completed. An internal *X•IPC* signal handler, within the user process, responds by performing the completion activity indicated for that operation (e.g., execute a user-specified callback function). The user process then returns to whatever it was doing before being interrupted.

The problems with employing signals for this purpose are the following:

- ❑ It is inherently difficult to program an application that may be interrupted by an operating system signal at almost any point in time. This is particularly true when working within a windowing environment such as X-Windows. Such environments are generally ill-behaved when user signaling is present.
- ❑ Many operating systems provide a means for waiting on multiple I/O related events, where each of the involved events is related to an open I/O descriptor. Employing signals as the method of *X•IPC* asynchronous notification precludes the possibility of multiplexing *X•IPC* events with operating system events. This essentially forces the developer who needs to block concurrently on I/O events *and* *X•IPC* events to do so separately, and in incompatible ways.

1.2 Using I/O Descriptors

X•IPC Version 3.0 uses a second and more generalized approach for *X•IPC* asynchronous event notification, that of employing file system I/O descriptors. Using I/O descriptors for notifying a process of asynchronous *X•IPC* events remedies the problems listed above. An explanation of the I/O descriptor approach follows.

The major difference between signal driven notification and I/O descriptor notification lies in how *X•IPC* internally notifies a process that an asynchronous *X•IPC* operation has completed. The I/O approach alerts the process by creating an I/O event on an I/O descriptor known to the process. Just how the process waits for and reacts to the I/O event (polling driven or interrupt driven) is left up to the application to decide.

An application can treat the *X•IPC* async I/O descriptor as it does any other I/O descriptor. It can set it to be blocking or non-blocking. It can additionally multiplex it with other descriptors.

This approach has the following advantages:

❑ An application's ability to react to asynchronous activity need not be signal driven. Applications having this requirement can be coded to poll the X•IPC asynchronous I/O descriptor at set (and safe) points within the application.

In addition, X-Windows applications can now be set to handle X•IPC asynchronous events as non-X-ToolKit events. Specifically, the `XtAddInput()` or `XtAppAddInput()` Xt library functions can be called to add the X•IPC I/O descriptor to the X-Window environment.

❑ The X•IPC Asynchronous I/O descriptor can be multiplexed with other I/O descriptors, so that waiting for X•IPC and non-X•IPC events can occur in a uniform manner.

1.3 Programming Concepts

Programming to use the X•IPC I/O descriptor method involves the following:

❑ The `XIPC_SETOPT_ASYNCFD` option.

❑ The `XipcAsyncIoDescriptor()` function.

❑ The `XipcAsyncEventHandler()` function.

1.3.1 THE XIPC_SETOPT_ASYNCFD OPTION

The default asynchronous mechanism used by X•IPC is the signaling method described above. Using the `XIPC_SETOPT_ASYNCFD` option directs X•IPC to use the I/O descriptor method instead. This option *must* be set before the process issues a call to the `XipcLogin()` API. Otherwise, the default (i.e., signal) approach is used.

1.3.2 THE XipcAsyncIoDescriptor() FUNCTION

A process that is using the I/O descriptor approach for handling its asynchronous X•IPC activity will inevitably need the value of the I/O descriptor being used. This value is returned by the `XipcAsyncIoDescriptor()` function call.

1.3.3 THE XipcAsyncEventHandler() FUNCTION

When a data-available event is sensed on the X•IPC I/O descriptor, an application must invoke the `XipcAsyncEventHandler()` function for actually processing the completed X•IPC operations. It is within this function that X•IPC executes the user-specified reaction to the operation's completion (e.g., execute a user-specified callback function).

1.3.4 USING PRIVATE QUEUES IN A THREADED APPLICATION

Private queues should be used in threaded applications so that:

♦ UNIX IPC queues will not fill up.

♦ Threads will not receive incorrect ACBs.

1.4 Examples

The following examples outline the programming steps necessary when using the I/O descriptor method of asynchronous operation notification.

1.4.1 AN EXAMPLE OF POLLING USING THE X♦IPC I/O DESCRIPTOR

The following program outline demonstrates how to poll the X♦IPC asynchronous I/O descriptor.

```
VOID
main()
{
    ASYNCRESULT     Acb;
    XINT            xipcfd;
    VOID            GotMessage();

    /*
     * Add XIPCASYNCIO to environment. It can be set to
     * any non-NULL value.
     */

XipcSetOpt(XIPC_SETOPT_ASYNCFD)

    /*
     * Login to an XIPC instance.
     */

    XipcLogin( ..., ... );

    /*
     * Get the XIPC aysnc I/O descriptor.
     */

    xipcfd = XipcAsyncIoDescriptor();

    /*
     * Issue an asynchronous XIPC operation. This example uses the
     * CALLBACK option. The POST or IGNORE option could have been
     * used as well.
     */

    QueReceive( ..., QUE_CALLBACK(GotMessage, &Acb));




    /*
     * Wait for a data-available event on the I/O descriptor.
     * This can be done either using the select() or poll()
     * system call. It can also involve other I/O descriptors.
     */

    select() or poll() xipcfd;

    /*
     * An XIPC asynchronous operation has completed.
     * Process it.
     */

    XipcAsyncEventHandler();

    ...
    ...
}

VOID
GotMessage(Acb)
ASYNCRESULT *Acb;
{
    if (Acb->Api.QueReceive.RetCode >= 0)
        printf("Got message: %s\n",  Acb->Api.QueReceive.MsgBuf);
}
```

1.4.2 AN X-WINDOWS EXAMPLE USING THE X♦IPC I/O DESCRIPTOR

The following program outline demonstrates how to use the X♦IPC asynchronous I/O descriptor within an X-Windows application.

```
VOID
main()
{
    ASYNCRESULT                 Acb;
    XINT                        xipcfd;
    VOID                        GotMessage();
    XtInputCallbackProc         MyXtCallBack();

    /*
     * Add XIPCASYNCIO to environment. It can be set to
     * any non-NULL value.
     */

XipcSetOpt(XIPC_SETOPT_ASYNCFD)
    /*
     * Login to an XIPC instance.
     */

    XipcLogin( ..., ... );

    /*
     * Get the XIPC aysnc I/O descriptor.
     */

    xipcfd = XipcAsyncIoDescriptor();

    /*
     * Register the xipcfd I/O descriptor with the X-ToolKit.
     * XtAppAddInput() could have been used as well.
     * The condition argument should be XtInputReadMask.
     */

    XtAddInput(xipcfd, XtInputReadMask, MyXtCallBack, NULL);

    /*
     * Issue an asynchronous XIPC operation. This example uses the
     * CALLBACK option. The POST or IGNORE option could have been
     * used as well.
     */

    QueReceive( ..., QUE_CALLBACK(GotMessage, &Acb));

    /*
     * Start the application.
     */
    XtMainLoop();
}

XtInputCallbackProc
MyXtCallBack(ClientData, Fd, XtId)
XtPointer       ClientData;
XINT            Fd;
XtInputId       XtId;
{
    /*
     * Process the XIPC event.
     */

    XipcAsyncEventHandler();
}


VOID
GotMessage(Acb)
ASYNCRESULT *Acb;
{
    if (Acb->Api.QueReceive.RetCode >= 0)
        printf("Got message: %s\n",     Acb->Api.QueReceive.MsgBuf);
}
```

2. MANUAL PAGES

The following pages describe the programming elements needed for using the I/O descriptor method of completion notification. These pages are also provided in the appropriate sections of the XIPC Reference Guide.

2.1 The XIPCASYNCIO Environment Variable

NAME

`XIPCASYNCIO` - The Asynchronous I/O Descriptor Environment Variable

DESCRIPTION

Setting the `XIPCASYNCIO` to any non-`NULL` value directs X•IPC to establish the process's X•IPC asynchronous notification mechanism to use an I/O descriptor instead of a signal.
The environment variable must be set at the time that the process issues an `XipcLogin()` function call, in order for the environment variable to have its effect. Otherwise, the default (i.e. signal) mechanism is set up.

FUNCTIONS REFERENCING "XIPCASYNCIO"

`XipcLogin()`, `XipcAsyncIoDescriptor()`,
`XipcAsyncEventHandler()`

2.2 The XipcAsyncIoDescriptor() Function

NAME

XipcAsyncIoDescriptor() - Access the Value of the XIPC Asynchronous I/O Descriptor

SYNTAX

```
#include "xipc.h"


XINT
XipcAsyncIoDescriptor()
```

PARAMETERS

None.

RETURNS

Value	**Description**
RC >= 0	Value of the XIPC asynchronous I/O descriptor.
RC < 0	Error (see error codes below).

DESCRIPTION

XipcAsyncIoDescriptor() returns the value of the I/O descriptor being used by XIPC for notifying the completion of asynchronous XIPC operations initiated by the calling process. This I/O descriptor is then typically used by the process for polling on, or for multiplexing along with, other I/O descriptors. Completion notification of an XIPC asynchronous operation is indicated as a data-available event on the I/O descriptor. The process should react by running the XipcAsyncEventHandler() function. This function processes the completing asynchronous XIPC operations.
The I/O descriptor may be integrated within an application's X-Window event loop environment. This is typically accomplished by passing the I/O descriptor to the XtAddInput() ot XtAppAddInput() Xt library function. The application must then be coded to call XipcAsyncEventHandler() at some point within the Xt callback function associated with the I/O event.

ERRORS

Code	**Description**
`XIPC_ER_NOACCESS`	Process not using XIPC asynchronous I/O descriptor method.
`XIPC_ER_SYSERR`	An internal error has occurred while processing the request.

2.3 The XipcAsyncEventHandler() Function

NAME

XipcAsyncEventHandler() - Process Completing X•IPC Asynchronous Operations

SYNTAX

```
#include "xipc.h"


XINT
XipcAsyncEventHandler()
```

PARAMETERS

None.

RETURNS

Value	Description
RC >= 0	Success.
RC < 0	Error (see error codes below).

DESCRIPTION

XipcAsyncEventHandler() processes completing asynchronous X•IPC operations and reads all data on the X•IPC async I/O descriptor. The function should be executed when a process is notified that one of its asynchronous X•IPC operations is complete. This generally occurs following the occurrence of a "data ready" event on the X•IPC asynchronous I/O descriptor.

The call to XipcAsyncEventHandler() may be placed within the main-line logic, within a signal handler or within an X-Windows event handler.

Note that XipcAsyncEventHandler() blocks if called when there are no outstanding AEBs; therefore, don’t call this function until the `select()` call returns, indicating "data ready." Refer to the previous X-Windows example for a program outline.

The XipcAsyncEventHandler() function should only be used when the process has chosen the I/O descriptor method of asynchronous notification by setting the `XIPCASYNCIO` environment variable.

ERRORS

Code	Description
`XIPC_ER_NOACCESS`	Process not using X•IPC asynchronous I/O descriptor method.
`XIPC_ER_SYSERR`	An internal error has occurred while processing the request.

USING EVENT OBJECTS FOR ASYNCHRONOUS OPERATIONS ON *WINDOWS NT/WINDOWS 95*

TABLE OF CONTENTS

1. *X•IPC* ASYNCHRONOUS OPERATIONS

A major advantage of developing multitasking and distributed applications using *X•IPC* is that it provides a rich set of asynchronous functionality. The benefits of such mechanisms are many, the most significant being that they allow an application's distributed processes to execute *concurrently* on a single multitasking platform as well as on multiple network nodes, thus leveraging the inherent parallelism provided by such environments.
A key step in the asynchronous execution of *X•IPC* operations is that of *completion notification*. This is the step by which *X•IPC* notifies a process of the completion of its asynchronous *X•IPC* operations.

1.1 Using Event Objects

X•IPC introduces a generalized approach for *X•IPC* asynchronous event notification—the use of Event Objects. An application can treat the *X•IPC* async event object as it does any other Windows NT/Windows 95 object. *X•IPC* sets the event object to non-signaled when an *X•IPC* function returns before the operation has completed. When the operation is completed, *X•IPC* sets the state as signaled. The thread can detect the state of the object by specifying the handle of the event object returned by the `XipcAsyncEventObject` function in one of the following Windows NT/Windows 95 functions: `WaitForSingleObject` or `WaitForMultipleObjects`.
The *X•IPC* asynchronous notification event handle is maintained on a per-thread basis. A thread should call `XipcAsyncEventHandler` only when it finds its own event object in a signaled state. If a single thread wants to wait on event objects of different threads, it can do so, but it should notify the owner of the *X•IPC* event by some other means of IPC so that the thread can call `XipcAsyncEventHandler`.

1.2 Programming Concepts

Programming to use the *X•IPC* Event Object involves the following:

- ❏ The `XipcAsyncEventObject()` function.
- ❏ The `XipcAsyncEventHandler()` function.

1.2.1 THE XipcAsyncEventObject() FUNCTION

A process that is using the event object approach for handling its asynchronous *X•IPC* activity will inevitably need the value of the event object being used. This value is returned by the `XipcAsyncEventObject` function call.

1.2.2 THE XipcAsyncEventHandler() FUNCTION

When a data-available event is sensed on the *X•IPC* event object, an application must invoke the `XipcAsyncEventHandler` function for actually processing the completed *X•IPC* operations. It is within this function that *X•IPC* executes the user-specified reaction to the operation's completion (e.g., execute a user-specified callback function).

1.3 Examples

The following example outlines the programming steps necessary when using the Event Object method of asynchronous operation notification.

1.3.1 AN EXAMPLE OF POLLING USING THE X♦IPC EVENT OBJECTS

The following program outline demonstrates how to poll the X♦IPC asynchronous event object.

```
VOID
main()
{
    ASYNCRESULT     Acb;
    HANDLE          hAsyncNotify;
    VOID            GotMessage();

    /*
     * Login to an XIPC instance.
     */

    XipcLogin( ..., ... );

    /* Get the XIPC aysnc notification handle.*/

    hAsyncNotify = (HANDLE)XipcAsyncEventObject();

    /*
     * Issue an asynchronous XIPC operation. This example uses the
     * CALLBACK option. The POST or IGNORE option could have been
     * used as well.
     */

    QueReceive( ..., QUE_CALLBACK(GotMessage, &Acb));

    /*
     * Wait for a data-available event.
     * Wait for either single objects or multiple objects ...
    */

     WaitForSingleObject(hAsyncNotify, INFINITE);
    /*
     * An XIPC asynchronous operation has completed.
     * Process it.
     */

    XipcAsyncEventHandler();

    ...
    ...
}


VOID
GotMessage(Acb)
ASYNCRESULT *Acb;
{
    if (Acb->Api.QueReceive.RetCode >= 0)
        printf("Got message: %s\n",  Acb->Api.QueReceive.MsgBuf);
}
```

1.4 The XipcAsyncEventObject() Function

NAME

`XipcAsyncEventObject()` - Access the Handle of the *XIPC* Asynchronous Event Object

SYNTAX

```
#include "xipc.h"


HANDLE
XipcAsyncEventObject()
```

PARAMETERS

None.

RETURNS

Value	Description
RC >= 0	Success
RC < 0	Failure

DESCRIPTION

XipcAsyncEventObject() returns the handle of the event object being used by *XIPC* for notifying the completion of asynchronous *XIPC* operations initiated by the calling process. This event object is then typically used by the process for polling on or for multiplexing along with other event objects.

ERRORS

Code	Description
`XIPC_ER_NOACCESS`	Process not using *XIPC* asynchronous I/O descriptor method.
`XIPC_ER_SYSERR`	An internal error has occurred while processing the request.

1.5 The XipcAsyncEventHandler() Function

NAME

`XipcAsyncEventHandler()` - Process Completing *X•IPC* Asynchronous Operations

SYNTAX

```
#include "xipc.h"


XINT
XipcAsyncEventHandler()
```

PARAMETERS

None.

RETURNS

Value	Description
RC >= 0	Success.
RC < 0	Error (see error codes below).

DESCRIPTION

XipcAsyncEventHandler() processes completing asynchronous *X•IPC* operations.
The function should be executed when a process has determined that one of its asynchronous *X•IPC* operations is complete. This determination is typically accomplished via a prior call to WaitForSingleObject(), as described in the earlier polling example.
Note that XipcAsyncEventHandler() blocks if called when there are no outstanding AEBs; therefore, don't call this function until async operations are complete and ready to be handled.

ERRORS

Code	Description
`XIPC_ER_NOACCESS`	User not logged in.
`XIPC_ER_SYSERR`	An internal error has occurred while processing the request.