CICS Transaction Gateway

Programming Guide

Version 7.1

SC34-6758-02

CICS Transaction Gateway

Programming Guide

Version 7.1

SC34-6758-02

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 239.

Third Edition (July 2008)

This edition applies to Version 7.1 of CICS Transaction Gateway, program number 5724-I81, 5655-R25 and 5724-J09. It will also apply to all subsequent versions, releases, and modifications until otherwise indicated in new editions.

This edition replaces SC34-6673. Technical changes to the text are indicated by a vertical line to the left of the change.

Contents

About this book

This information is an introduction to programming for the CICS® Transaction Gateway and the CICS Universal Client. It provides the information that you need to enable your user applications to interact with CICS server applications.

The CICS Universal Client and CICS Transaction Gateway provide secure, easy access to multiple CICS servers and support user applications developed in C++, C, COBOL and COM based programming languages.

In addition the CICS Transaction Gateway supports remote access to CICS servers from user applications developed in Java™. It also provides support for the J2EE Connector Architecture (JCA) allowing user applications to be deployed into WebSphere® Application Server.

This book starts by describing the application programming interfaces (APIs) in a way that is independent of programming languages. It then gives guidance on programming in the supported languages.

Installation path

The term <install_path> is used in file paths to represent the directory where you installed the product. See the *CICS Transaction Gateway: Administration* book for your operating system, for the default installation locations.

Directory delimiters

References to directory path names in this book use the Microsoft® Windows® convention of a backslash (\) as delimiter, instead of the forward slash (/) delimiter used on UNIX® and Linux® operating systems.

Information specific to your operating system

Unless otherwise specified, the term *Windows* refers to Windows 2000, Windows 2003, Windows XP, and Windows Vista.

The term *Windows Terminal Server* means a server with the *Terminal Services* feature enabled.

What's new in programming

- CICS Transaction Gateway V7.1 introduces IP interconnectivity (IPIC) which provides enhanced TCP/IP support for the J2EE Connector Architecture (JCA). This means that JCA now supports channels and containers, using a MappedRecord structure to hold your data. When the MappedRecord is passed to the execute() method of ECIInteraction, the method uses the MappedRecord itself to create a channel and converts the entries inside the MappedRecord into containers before passing them to CICS. For more information on channels and containers, see "Introduction to channels and containers" on page 84.
- CICS Transaction Gateway enables users and third party vendors to write tools to access request specific information. User exit points are provided at key points in the product to allow user code to be run within the context of each individual transaction. For further information see, Chapter 11, "Request monitoring user exits," on page 173.
- On distributed platforms, you can use ctgadmin to control request monitoring user exits dynamically. The eventFired() method can now be called with a new RequestEvent of "Command".
- You can use the statistics API in remote mode.

Chapter 1. Overview

This information gives an overview of programming for the CICS Transaction Gateway. Topics covered include APIs, ancillary functions, user applications, and supported programming languages.

Application Programming Interfaces

The CICS Transaction Gateway and CICS Universal Client support the integration of CICS systems and client systems. They have a standard set of functions to allow user applications to call CICS programs or initiate CICS 3270 transactions.

Three Application Programming Interfaces (APIs) are available to enable user applications to access and update CICS facilities and data. These are the External Call Interface (ECI), the External Presentation Interface (EPI) and the External Security Interface (ESI).

External Call Interface (ECI)

A user application can call a CICS program in a CICS server by using an ECI request. The user application can connect to several CICS servers at the same time and have several called CICS programs running concurrently. The CICS programs must be COMMAREA-based programs that can be called using the CICS command

CICS programs that are invoked by an ECI request must follow the rules for distributed program link (DPL) requests. For information on DPL requests see the *CICS Application Programming Guide*. For information on the API restriction for DPL requests refer to Appendix G of the *CICS Application Programming Reference*.

External Presentation Interface (EPI)

A user application can install and delete virtual IBM® 3270 terminals in CICS servers by using the EPI. The 3270 terminal definitions used by the EPI are treated by CICS servers as remote 3270 terminal definitions and therefore support automatic transaction initiation requests (ATI). For more information on ATI see *CICS Application Programming Guide*.

External Security Interface (ESI)

A user application can perform certain security functions by using an ESI request. This includes accessing the information about user IDs held in the CICS External Security Manager (ESM), and setting the default security credentials to be used for a server connection.

Statistical data API

A user application can collect statistical information about a running CICS Transaction Gateway.

Ancillary functions

Several ancillary functions are provided with the CICS Transaction Gateway and CICS Universal Client:

List CICS systems

To determine which CICS servers ECI and EPI requests can be directed to, user applications can query the CICS Transaction Gateway or CICS Universal Client for a list of CICS systems. The query returns a list of the CICS servers that have been defined within the CICS Transaction Gateway or CICS Universal Client. There is no guarantee that communication links exist between the CICS servers and the CICS Transaction Gateway or CICS Universal Client, or that any of the CICS servers are actually available.

Code page information

When using the application programming interfaces of the CICS Transaction Gateway or CICS Universal Client to invoke CICS programs, data conversion is an important consideration.

If the code page of the user application is different from the code page of the CICS server, or the byte order of binary data is in a different format, you might need to convert the data in a COMMAREA or container. You can do this conversion by making use of CICS supplied data conversion capabilities on the CICS server, provided by the DFHCCNV program and controlled by the DFHCNV macro definitions. In this case all data conversion is performed on the CICS server. Alternatively, you can make use of data marshalling utilities provided within your user application development environment.

If you are using Java you can determine the code page of the Client daemon from the user application. For more information about this utility refer to the Javadoc supplied with the product.

RACF user ID certificate mapping

The CICS Transaction Gateway for z/OS® provides a java class that can be used to map an X.509 certificate to a RACF® user ID. For more information about this utility refer to “CICS Transaction Gateway security classes” on page 76.

BMS map conversion utilities

The CICS Transaction Gateway provides utilities to allow CICS BMS map definitions to be imported, and the resulting information used by EPI-based user applications to access BMS map data as named fields. Two separate utilities are provided, one for use with the Java EPI support classes and the other for the C++ classes.

User applications

The CICS Transaction Gateway for z/OS supports only Java Client applications. On other platforms the CICS Transaction Gateway supports both Client applications and Java Client applications. The CICS Universal Client supports only Client applications.

Common capabilities

Client applications and Java Client applications have the following capabilities in common:

- They can be written to access one or more CICS servers.
- They can connect to several CICS servers at the same time.
- They can have several program calls running concurrently.
- The same user application can be written to include any combination of ECI, EPI or ESI requests.

Client applications

Client applications run locally on the machine where the CICS Transaction Gateway or CICS Universal Client has been installed. They enable access to CICS server transactions and programs from the host machine, as shown in Figure 1 below and Figure 2 on page 5 of the Java Client applications.

Client applications communicate with CICS servers using the Client API. The Client daemon processes any ECI, EPI and ESI requests, sending and receiving the appropriate flows to and from the CICS servers to satisfy these Client application requests.

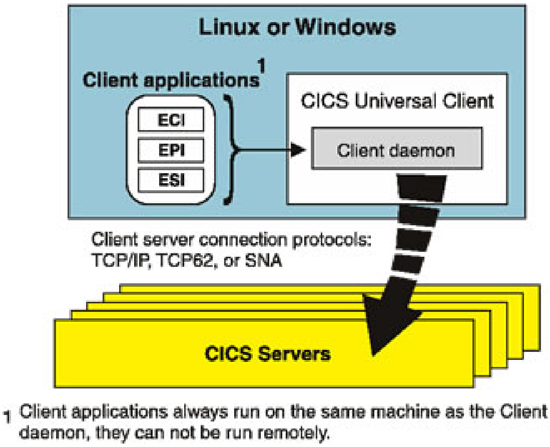

Linux or Windows
Client applications 1
ECI
EPI
ESI
CICS Universal Client
Client daemon
Client server connection protocols:
TCP/IP, TCP62, or SNA
CICS Servers
1 Client applications always run on the same machine as the Client daemon, they can not be run remotely.

Figure 1. CICS Universal Client for Linux or Windows

Java Client applications

Java Client applications are written in Java and include servlets, enterprise beans and applets. They use the Gateway classes to communicate with CICS servers. Java Client applications run in local or remote mode. The Gateway classes provide access to CICS server transactions and programs for large numbers of concurrent users.

Figure 2 on page 5 shows Java Client applications running in both local and remote mode on a UNIX, Linux or Windows System.

Note: Java applet support is provided for compatibility with previous versions of the CICS Transaction Gateway but it is recommended that users migrate to a JCA based solution. Java applet support may be removed in a future release of CICS Transaction Gateway.

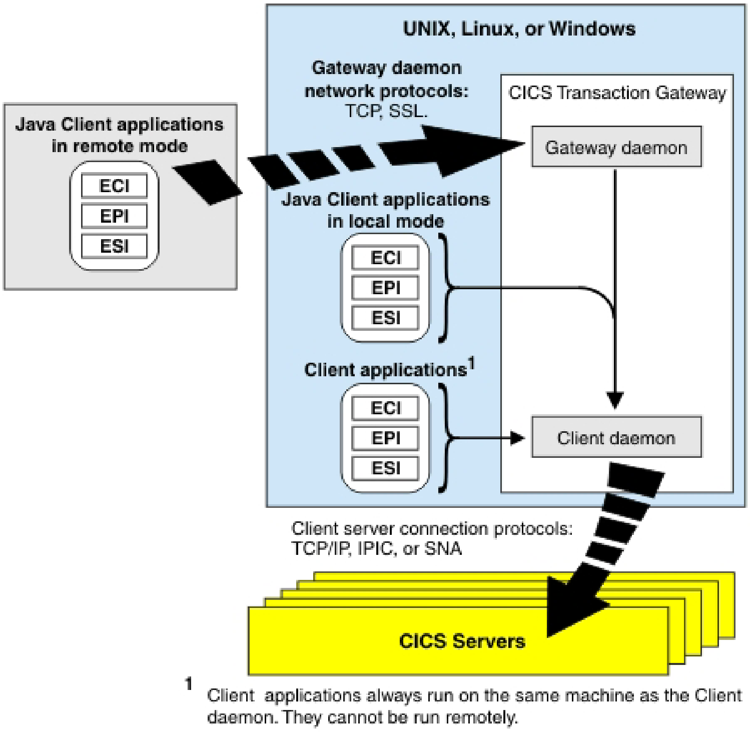

UNIX, Linux, or Windows
Gateway daemon network protocols: TCP, SSL.
CICS Transaction Gateway
Java Client applications in remote mode
ECI
EPI
ESI
Gateway daemon
Java Client applications in local mode
ECI
EPI
ESI
Client applications1
ECI
EPI
ESI
Client daemon
Client server connection protocols: TCP/IP, IPIC, or SNA
CICS Servers
1 Client applications always run on the same machine as the Client daemon. They cannot be run remotely.

Figure 2. CICS Transaction Gateway for UNIX, Linux, or Windows

Figure 3 on page 6 shows Java Client applications running in both local and remote mode on a z/OS system.

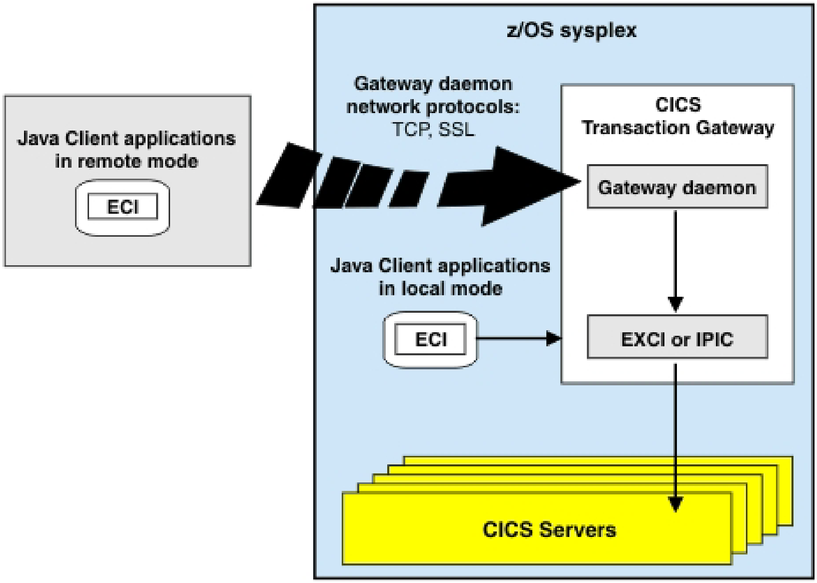

z/OS sysplex
Java Client applications in remote mode
ECI
Gateway daemon network protocols: TCP, SSL
CICS Transaction Gateway
Gateway daemon
Java Client applications in local mode
ECI
EXCI or IPIC
CICS Servers

Figure 3. CICS Transaction Gateway for z/OS

J2EE Connector Architecture (JCA) applications

The CICS Transaction Gateway implements the JCA by providing J2EE CICS resource adapters. These resource adapters support the J2EE Common Client Interface (CCI) defined by the JCA and are a middle-tier between JCA compliant applications and the CICS Transaction Gateway. The J2EE application server can be run locally on the same machine as the CICS Transaction Gateway, or remotely as shown in Figure 4 on page 7.

JCA compliant applications can be developed and deployed in a managed or nonmanaged environment. In a managed environment, JCA applications can exploit the quality of service provided by the J2EE application server.

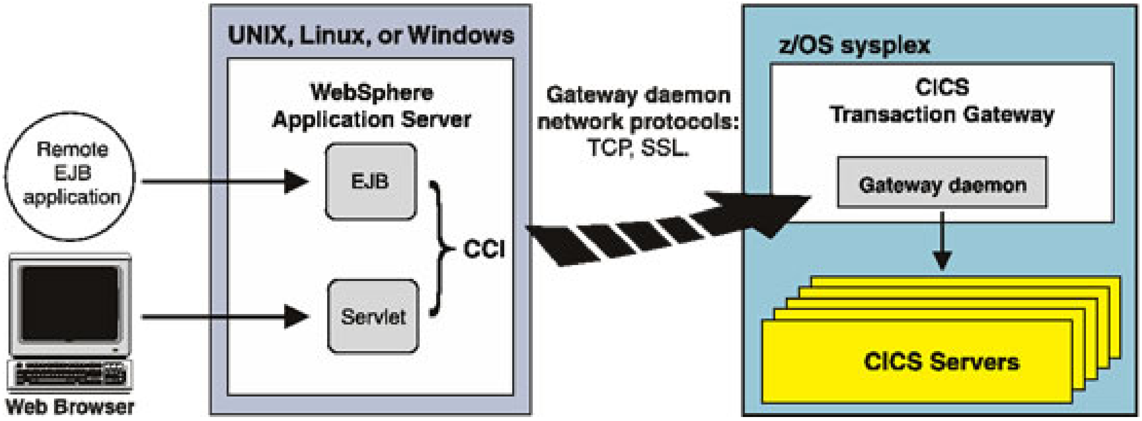


Figure 4. CICS Transaction Gateway with WebSphere Application Server in remote mode

Supported programming languages

Table 1 shows which programming languages are supported by the CICS Universal Client for each platform and application programming interface (API).

Table 1. Supported programming languages of the CICS Universal Client

Platform and API	C	C++	COBOL	COM
ECI for Windows	●	●	●	●
EPI for Windows	●	●	●	●
ESI for Windows	●	●	●	●
ECI for Linux	●	●	○	○
EPI for Linux	●	●	○	○
ESI for Linux	●	●	○	○

ECI — External Call Interface
EPI — External Presentation Interface
ESI — External Security Interface
● — supported ○ — not supported

Note: ESI requests are currently only available when the server connections have been configured to use the SNA network protocol, and the configured CICS server supports Password Expiration Management (PEM).

For information on supported compilers and application development tools see the *CICS Transaction Gateway: Administration* book for your operating system.

Table 2 shows which programming languages are supported by the CICS Transaction Gateway for each platform and application programming interface (API).

Table 2. Supported programming languages of the CICS Transaction Gateway

Platform and API	C	C++	COBOL	COM	Java Support Classes	Java Base Classes	JCA
ECI for Windows	●	●	●	●	○	●	●
EPI for Windows	●	●	●	●	●	●	●
ESI for Windows	●	●	●	●	○	●	○
ECI for UNIX and Linux	●	●	○	○	○	●	●
EPI for UNIX and Linux	●	●	○	○	●	●	●
ESI for UNIX and Linux	●	●	○	○	○	●	○
ECI for z/OS	○	○	○	○	○	●	●
EPI for z/OS	○	○	○	○	○	○	○
ESI for z/OS	○	○	○	○	○	○	○
ECI — External Call Interface EPI — External Presentation Interface ESI — External Security Interface ● — supported ○ — not supported							

Note: ESI requests are currently only available when the server connections have been configured to use the SNA network protocol, and the configured CICS server supports Password Expiration Management (PEM).

Chapter 2. External Call Interface (ECI)

This information describes the basic functions of the External Call Interface (ECI).

The ECI request

External calls to CICS

An ECI request calls a CICS program on a CICS server. This is known as making an external call to CICS and is the primary purpose of the ECI request. If no CICS server is selected, the default CICS server is used.

The ECI request can make four different types of call:

- Program link calls
- Status information calls
- Reply solicitation calls
- Callbacks

The following sections describe these external calls to CICS.

Input and output information for external calls to CICS

The following input parameters can be passed to the CICS server with an ECI call:

CHANNEL
A communication area used for passing containers to a server program.

COMMAREA
A communication area used for passing input to a server program.

ECI timeout
The maximum wait time for a response to an ECI request.

LUW control
The way in which a Logical Unit of Work (LUW) is started, continued and ended.

LUW identifier
A token which identifies the ECI call as part of a LUW.

Password
The password provided for security checking on an ECI call.

Program name
The name of a program to be run on a CICS server.

Server name
The name of the CICS server that the ECI call is directed to.

TPNName
The transaction ID of the CICS mirror program.

TranName
The transaction ID seen in the exec interface block (EIB) by the CICS mirror program.

Userid
The user ID provided for security checking on an ECI call.

The following output can be returned to the user application following an ECI call.

Abend code
The code returned when a server program has ended abnormally.

CHANNEL
A communication area that holds containers passed from a server program.

COMMAREA
The communication area that contains output from a server program.

LUW identifier
A token which identifies the ECI call as part of a LUW.

Program link calls

Program link calls cause the CICS mirror transaction to be attached to run a server program on the CICS server.

ECI request program link calls can be synchronous or asynchronous:

Synchronous
Synchronous calls are blocking calls. The user application is suspended until the called server program has finished and a reply is received from CICS. The received reply is immediately available.

Asynchronous
Asynchronous calls are nonblocking calls. The user application gets control back without waiting for the called server program to finish. The reply from CICS can be retrieved later using one of the reply solicitation calls or a callback. See "Retrieving replies from asynchronous ECI requests" on page 12. An asynchronous program link call is outstanding until a reply solicitation call, or the callback, has retrieved the reply.

Synchronous and asynchronous program link calls can be nonextended or extended:

Nonextended
The CICS server program, not the user application, controls whether recoverable resources are committed or backed out. Each program link call corresponds to one CICS transaction.

Extended

The user application controls whether recoverable resources are committed or rolled back. Multiple calls are possible, allowing a LUW to be extended across successive ECI requests to the same CICS server. This is known as an *extended logical unit of work* (extended LUW).

CICS user applications are often concerned with updating recoverable resources. A LUW is the processing that a CICS server program performs between syncpoints. A syncpoint is the point at which all changes to recoverable resources that were made by a task since its last syncpoint are committed. LUW management is performed by the user application, using the *commit* and *rollback* functions:

Commit

Ends the current LUW and any changes made to recoverable resources are committed.

Rollback

Terminates the current LUW and backs out (rolls back) any changes made to recoverable resources since the previous syncpoint.

ECI-based communications between the CICS server and the CICS Transaction Gateway or CICS Universal Client are known as conversations. A nonextended program link ECI call is one conversation. A series of extended ECI calls followed by a commit or rollback is one conversation.

On platforms other than z/OS, a given logical unit of work can include ECI requests to only one server. Only one transaction can be active at a time in a logical unit of work, so care must be taken with non-synchronous requests.

Managing Logical Units of Work

On a successful return from the first of a sequence of extended ECI calls for a LUW, the user application is returned a LUW identifier corresponding to an instance of a CICS mirror transaction. Specifying this LUW identifier in subsequent ECI calls means that these calls will be processed by the same CICS mirror transaction. All program link calls for the same LUW are sent to the same server.

When the user application makes an ECI commit or rollback call, the CICS server attempts to commit or back out changes to recoverable resources. The user application is advised whether or not the attempt was successful. If a LUW is outstanding (incomplete), the user application should issue an extended ECI commit or rollback call to the CICS server. If the execution of a user application completes without committing or rolling back an outstanding LUW, the CICS Transaction Gateway or CICS Universal Client attempts to back out the LUW.

If an extended ECI call fails, the user application must check if a nonzero LUW identifier was returned. If so, this indicates that the LUW is still outstanding and should be committed or rolled back. If not, the problem is a lost communications link with the CICS server.

An ECI user application using an extended LUW might cause other user applications to be suspended waiting for CICS resources, which are held for the duration of the LUW.

Status information calls

Status information calls retrieve status information about the connection between the client and server systems.

The status of connected servers is updated as a result of requests being flowed and protocol specific events. The status returned is the last known state of connected servers, which might not be the same as the current state.

ECI request status link calls can be synchronous or asynchronous.

There are three types of status information call:

Immediate
Requests status information to be sent to the user application immediately it becomes available.

Change
Requests status information to be sent to the user application when the status changes from some specified value. Change calls are always asynchronous.

Cancel
Cancels an earlier **change** call.

Retrieving replies from asynchronous ECI requests

Callbacks

Callbacks enable the CICS server to drive specific function provided by the user application when an asynchronous program link call completes. This is the recommended way of handling replies from ECI requests.

Reply solicitation calls

Reply solicitation calls retrieve the reply for an asynchronous call. A user application that issues asynchronous calls can have several ECI requests outstanding at any time. It is the responsibility of the calling application to solicit the reply for an ECI request. If no reply is available, reply solicitation calls can either wait for a reply or return control directly to the user application. There are two types of reply solicitation call:

General
Retrieves any reply for any outstanding ECI request.

Specific
Retrieves a reply for a specific ECI request. A unique message qualifier is used to identify the reply for that request. It is the responsibility of the programmer to assign different message qualifiers to different asynchronous calls within a single application.

ECI and CICS transaction IDs

For ECI calls the transaction ID of the mirror transaction can be controlled via two different parameters:

- TPNName
- TranName

Specify TPNName to change the name of the CICS mirror transaction that the called program will run under. For example, you can specify TPNName if you need a transaction definition with different attributes from those defined for the default mirror transaction. This option is like the TRANSID option on an EXEC CICS LINK command. The transaction ID is available to the server program in the exec interface block (EIB). You must define a transaction on the CICS server for this transaction ID that points to the DFHMIRS program. Note that TPNName takes precedence if both TranName and TPNName are specified. If neither TPNName nor TranName is specified, the ECI Program Link call is attached to the default mirror transaction on the server. The default mirror transaction is CSMI if the CICS Transaction Gateway is on z/OS and CPMI in all other cases.

If TranName is specified, the called program runs under the default mirror transaction, but is linked to under the TranName transaction ID. This name is available to the called program in the (EIB) for querying the transaction ID.

Table 3 shows the name of the CICS mirror transaction and the name stored in EIBTRNID according to whether or not TPNName and TranName are specified.

Table 3. Specifying TPNName and TranName

TPNName specified	TranName specified	Mirror transaction name	Name in EIBTRNID
Y	Y	TPNName	TPNName
Y	N	TPNName	TPNName

Table 3. Specifying TPNName and TranName (continued)

N	Y	default	TranName
N	N	default	default

Timeout of the ECI request

An ECI timeout can occur either before or after the ECI request has been sent to the CICS server. An ECI timeout is the time the Client daemon will wait for a response to an ECI request sent to a CICS server. The ECI request has two timeout conditions, *request timeout* and *response timeout*.

The ECI timeout value is specified in seconds. Because the Client daemon rounds values under a second, some ECI requests might time out before the period specified in eci_timeout. For example if eci_timeout is set to 2, some ECI requests might time out in less than 2 seconds. If the ECI requests always have to wait for at least 2 seconds, set the timeout to 3.

Request timeout

A request timeout occurs before the request has been forwarded to the CICS server. The requested program was not called, and no server resources have been updated.

This can happen for the following reasons:

- The call was intended to start, or be the whole of, a new LUW. The LUW is not started, and no recoverable resources are updated.
- The call was intended to continue an existing LUW. The LUW continues, but no recoverable resources are updated, and the LUW is still uncommitted.
- The call was intended to end an existing LUW. The LUW continues, no recoverable resources are updated, and the LUW is still uncommitted.

Response timeout

A response timeout occurs after the request has been forwarded to the CICS server. It can happen to a synchronous call, an asynchronous call, or to the reply solicitation call that retrieves the reply from an asynchronous call.

This can happen for the following reasons:

- The call was intended to be the only call of a new LUW. The LUW was started, but the user application cannot determine whether updates were performed, and whether they were committed or backed out.
- The call was intended to end an existing LUW. The LUW has ended, but the user application cannot determine whether updates were performed, and whether they were committed or backed out.

- The call was intended to continue or to end an existing LUW. The LUW persists, and changes to recoverable resources are still pending.

Security in the ECI

The ECI uses conversation-level security based on the SNA LU 6.2 model.

ECI security involves:

Authentication
Checking that the user ID and password information associated with an ECI call is valid.

Authorization
Checking that the authenticated user is allowed access to the requested resource. This check is performed on the CICS server.

The user application can set the user ID and password on an ECI request for a conversation with a specific CICS server. These values override any default values set for the server connection. For information about how to set the connection userid and password, refer to the *CICS Transaction Gateway or CICS Universal Client: Administration* book for your operating system.

Chapter 3. External Presentation Interface (EPI)

This information describes the External Presentation Interface (EPI).

EPI concepts

EPI allows a user application program to access 3270–based transactions on one or more CICS servers. The user application can establish one or more resources and act as the operator, starting 3270-based CICS transactions and sending and receiving data associated with those transactions.

Adding and deleting terminals

EPI functions can be used to add terminals to CICS and delete them when they are no longer required. The user application that installs a terminal has exclusive use of that terminal until the terminal is deleted.

Adding a basic terminal to CICS is a synchronous operation. Adding an extended terminal can be synchronous or asynchronous. If the operation is synchronous, control is not returned to the user application until the install request has completed. If the operation is asynchronous, control is returned to the user application as soon as any parameters have been validated. Basic and extended terminals are described in “Terminal characteristics” on page 19.

Starting transactions

When a user application has added a terminal to a CICS server, the application can start a transaction from that terminal. To the CICS server it appears as if an operator has entered a transaction name at a terminal.

There are four ways in which you can start a transaction and associate data with it:

1. By supplying the transaction identifier and any transaction data.
2. By combining a transaction identifier and transaction data into a 3270 data stream, and supplying the data stream.
3. By using Automatic Transaction Initiation (ATI) to start a transaction. Some programming languages do not support ATI.
4. By specifying the **TRANSID** option on the **EXEC CICS RETURN** command in the CICS server program to indicate the next transaction to run. If you also specify the **IMMEDIATE** option, the next transaction is started without any intervention from the user application and regardless of any outstanding ATI requests for that terminal.

Sending and receiving data

When a transaction is running on CICS, data is passed between CICS and the user application. This might be data produced by the transaction or one or more messages from the CICS server, for example terminal error messages. If the data is in the form of BMS map data, CICS also supplies the map name and map set name. If the map is to be returned to CICS for further processing, the user application must also return the map name and map set name.

Some programming languages have APIs that provide functions to help process the data stream.

There are two different programming models for EPI-based applications:

- The screen model allows the user application to handle the 3270 data based on the structure of the fields in the 3270 data stream. In some languages it is also possible to import BMS map data to help with this process.
- With the 3270 model, the user application reads the 3270 data stream as a simple data record and is responsible for parsing the information that it contains.

The user application is responsible for presenting the data received. The application can present the data by emulating a 3270 terminal, or it might present a different view. For example:

- A Windows application might use the Windows graphical user interface.
- A Solaris application might use Open Look.

Managing CICS conversations

A *conversational transaction* is one which processes several sets of input from a terminal before returning control to CICS. The length of time required for a response from a terminal is much longer than the time taken to process it, therefore a conversational transaction lasts much longer than a *nonconversational transaction,* which processes one set of input before relinquishing control. While a transaction is running it is using storage and resources which might be needed by other transactions. For this reason many CICS transactions operate in pseudoconversational mode.

A *pseudoconversational transaction* is one in which the conversation between a terminal and a server is broken up into a number of segments, each of which is a nonconversational transaction. As each transaction ends, it provides the name of the transaction to be run to process the next input from the terminal. When a transaction that has just ended specifies the name of a transaction to process the next input, this name is passed to the user application. The application must not attempt to start a different transaction, but must use the returned information to start the specified transaction and send the data it is expecting.

Terminal characteristics

Most terminal attributes are supplied by the CICS server but some can optionally be determined by the user application. Terminals can be either basic or extended. Extended terminals have more attributes than basic terminals. An extended terminal can be purged while a transaction is running but basic terminals can only be deleted when they are in the idle state.

You can specify the following attributes as input parameters for both basic and extended terminals:

Model For autoinstalled terminals this is the name of an existing terminal definition on the CICS server which is to serve as a model for this terminal.

Server name
The name of the CICS server where the terminal is to be installed.

Netname
The network name of the terminal. .

The following additional attributes can be specified for extended terminals:

Code page
The code page used by the user application for data passed between the terminal resource and CICS transactions.

Install timeout
The maximum length of time that the CICS Transaction Gateway or CICS Universal Client will wait for a terminal to be installed on the selected CICS server. If not specified there is no limit to the wait time. Refer to “Timeout of the EPI request” on page 20 for more information.

Password
The password that is to be associated with the terminal for security checking.

Read timeout
The maximum length of time that the CICS Transaction Gateway or CICS Universal Client will wait for a response from the user application. If not specified there is no limit to the wait time. Refer to “Timeout of the EPI request” on page 20 for more information.

Sign-on capability
Whether the terminal is capable of running a CICS sign-on transaction. If not specified, the terminal has the default sign-on capability of the CICS server type. Sign-on capability and sign-on incapability are described in more detail in “Specifying terminal Sign-on Capability” on page 21.

Userid
The userid that is to be associated with the terminal for security checking.

The following attributes are returned to the user application by the CICS server when a terminal is added:

Color Whether the terminal supports color.

Columns
The number of columns supported by the terminal.

Error last line
Whether error messages are displayed on the last line of the terminal.

Error message color
The color used to display error messages on the terminal.

Error message highlight
The highlight value used to display error messages on the terminal.

Error message intensity
The intensity with which error messages are displayed on the terminal.

Extended highlight
Whether the terminal supports extended highlighting.

Maximum data
The maximum length of data that can be sent from and received by the terminal.

Netname
The network name of the terminal.

Rows The number of rows supported by the terminal.

Server name
The name of the CICS server where the terminal is installed.

Sign-on capability
The sign-on capability assigned to the terminal by the server.

Terminal ID
The terminal ID generated by CICS.

Timeout of the EPI request

There are two EPI timeout conditions, *install timeout* and *read timeout*.

Install timeout
Install timeout is the maximum length of time that the CICS Transaction Gateway or CICS Universal Client will wait for a terminal to be installed on a CICS server. If a response is not received from the server within the specified time, control is returned to the user application with an appropriate return code. If the Client daemon is subsequently notified that the terminal has been installed in the server, the Client daemon deletes the terminal. If no install timeout value is specified, there is no limit to the wait time.

Read timeout
Read timeout is the maximum length of time that the CICS Transaction Gateway or CICS Universal Client will wait for a response from the user application. This period of time starts when a user

application has received an EXEC CICS RECEIVE or CONVERSE command issued by CICS. A read timeout occurs if no data is returned before the period specified has elapsed. If no read timeout value is specified, there is no limit to the wait time. When a read timeout occurs, the transaction on the CICS server is terminated abnormally.

Security in the EPI

A userid and password might be required for each conversation that takes place between the CICS Transaction Gateway or CICS Universal Client and the CICS server, depending on how the CICS Transaction Gateway or CICS Universal Client and the CICS server have been configured.

EPI security involves:

Authentication
The CICS server checks that the userid and password information associated with a terminal is valid. The frequency with which the userid and password are authenticated by the CICS server depends on whether the terminal is sign-on capable or sign-on incapable.

Authorization
The CICS server checks that the terminal is allowed access to the requested resource.

The userid and password can be set at terminal or connection level. Both types can be set by the user application. If there are no userid and password values for the terminal, the values for the connection are used. For information about how to set the connection userid and password, refer to the *CICS Transaction Gateway: Administration* book for your operating system. The requirement for a userid and password depends on the CICS server configuration.

Specifying terminal Sign-on Capability

Sign-on capability is one of the attributes that can be specified for extended terminals. A request to change sign-on capability is effective only for z/OS CICS servers. For other server types and for basic terminals, sign-on capability depends on the default for the CICS server type. The sign-on capability of a terminal is returned to the user application in the sign-on capability field of the terminal details. Table 4 on page 22 shows the results of a request to override the default sign-on capability for different CICS servers.

Table 4. Specifying the sign-on capability attribute for different servers

Request	**Resulting sign-on capability of terminal**			**Value of sign-on capability in terminal details**		
	CICS Transaction Server for z/OS	**CICS Transaction Server for iSeries®**	**TXSeries® and CICS Transaction Server for Windows**	**CICS Transaction Server for z/OS**	**CICS Transaction Server for iSeries**	**TXSeries and CICS Transaction Server for Windows**
sign-on capable	sign-on capable	sign-on incapable	sign-on capable	sign-on capable	sign-on unknown	sign-on unknown
sign-on incapable	sign-on incapable	sign-on incapable	sign-on capable	sign-on incapable	sign-on unknown	sign-on unknown

The following sections describe sign-on incapable and sign-on capable terminals.

Sign-on incapable terminals

Sign-on incapable terminals do not allow sign-on transactions to be run. When a terminal is sign-on incapable, the userid and password must be passed to the CICS server if the connection is configured with ATTACHSEC=IDENTIFY, and are then authenticated for every transaction started against that terminal. The transaction is executed in the server with the authorities assigned to the authenticated userid.

The userid and password for an extended terminal can be specified by a user application when a terminal is added.

The user application can change the security settings of an extended terminal at any time. The new settings will be used when further transactions are started for the terminal.

The user application can also set a default userid and password to be used with a particular CICS server. For details, refer to the *CICS Transaction Gateway or CICS Universal Client: Administration* guide for the operating system that you are using.

Sign-on capable terminals

Sign-on capable terminals allow CICS-supplied (CESN), or user-written sign-on transactions to be run. When a terminal is sign-on capable it is the responsibility of the user application to start the sign-on transaction. The userid and password are determined by the user application and are embedded in the 3270 data. If the userid is authenticated, subsequent transactions started at the terminal are executed in the CICS server with the authorities assigned to the authenticated userid. Transactions started before a sign-on transaction has completed have the authorities granted to the default userid defined for the CICS server. A check is also done against the userid associated with the connection to see whether the CICS Transaction Gateway or CICS Universal Client has authority to execute the transaction.

The user application can start a signoff transaction at the terminal. The user can also be signed off by the server following a predefined period of inactivity. The user application should allow for this possibility. In either case, subsequent transactions started at the terminal are executed with the authorities assigned to the CICS server default userid.

For transactions attempting to access resources, security checking is done against the userid associated with the connection and the signed-on user's userid.

Automatic transaction initiation (ATI)

ATI is the CICS process that allows a transaction to be scheduled against a specified terminal.

An ATI request from an EPI user application can cause the scheduled initiation of a transaction in a CICS server against any EPI installed terminal.

Either the user application or the CICS systems administrator can enable or disable automatic initiation of transactions for a terminal . The default state is disabled. If ATI requests are enabled and an ATI request is issued in the CICS server, the request is started when the terminal is idle. Any ATI requests issued while ATI requests are disabled are queued, and started when ATI requests are next enabled. ATI requests for a terminal are queued while a transaction is in progress on that terminal.

Restrictions on application design when using EPI

A CICS transaction that sends data to an EPI user application cannot:

- Use 14- and 16- bit addresses and structured fields, as the CICS Transaction Gateway and CICS Universal Client support only the ASCII-7 subset of the 3270 data stream architecture. Only 12-bit SBA addressing is supported. Consequently, the maximum screen size for EPI terminals is 27 rows by 132 columns.
- Use the purge function to cancel ATI requests queued against the terminal. If a CICS transaction uses EXEC CICS START with the DELAY option to schedule transactions to a terminal resource autoinstalled by a user application, the user application should ensure that delayed ATI requests are not lost when the terminal resource is deleted. See your server

documentation to determine the effects of deleting a terminal resource when delayed ATI requests are outstanding.

An EPI user application cannot:

- Use basic mapping support (BMS) paging.
- Determine the alternate screen size of the terminal resource definition, although it can determine the default screen size.

An EPI user application communicating with CICS Transaction Server for iSeries cannot:

- Support languages that use DBCS.
- Support sign-on capable terminals.
- Start the CEDA transaction from a client terminal.
- Use PF1 to get CICS online help from a client terminal.

3270 data streams for the EPI

The data streams implemented for the EPI follow those defined in the *3270 Data Stream Programmer's Reference*. All data flows for the EPI are in ASCII format, and structured fields are not supported. Data flows are defined under the following topics in the *3270 Data Stream Programmer's Reference*:

- Introduction to the 3270 data stream (excluding structured fields)
- 3270 data stream commands
- Character sets, orders and attributes
- Keyboard and printer operations.

Be aware that the contents of the data buffer may be code-page converted if the buffer is passed between CICS systems, in which case the data should be limited to ASCII and EBCDIC characters.

If a CICS transaction issues EXEC CICS SEND MAP and EXEC CICS RECEIVE MAP commands, CICS converts the data from the BMS structure to a 3270 data stream. In this case, the application receives 3270 data from CICS and should return valid 3270 data to be converted for the transaction.

Chapter 4. External Security Interface (ESI)

This information describes the External Security Interface (ESI).

ESI functions

The ESI allows a user application to invoke password management functions on an attached CICS server.

Input and output information for ESI functions

The following input parameters can be specified for an ESI function:

New password
The new password for the specified user.

Current password
The current password for the specified user.

Password
The password to be set or verified for the specified user

System
The name of a CICS server containing the user whose password is to be set, changed, or verified. If this value is not specified the default server is selected.

User ID
The ID of the user whose password is to be set, changed, or verified.

The following output parameters can be returned from an ESI function:

Expiry date
The date on which the password will expire.

Expiry time
The time at which the password will expire.

Invalid count
The number of times an invalid password has been entered for the specified user.

Last access date
The date on which the userid was last accessed.

Last access time
The time at which the userid was last accessed.

Last verify date
The date on which the password was last verified.

Last verify time
The time at which the password was last verified.

Using ESI to manage passwords

ESI provides a security management API which can be used to manage the user IDs and passwords that the ECI and EPI use.

The user application can perform the following functions:

- Verify that a password matches the password recorded by the CICS External Security Manager (ESM) for a specified user ID.
- Change the password recorded by the CICS ESM for a specified user ID.
- Determine if a user ID is revoked, or a password has expired.
- Obtain additional information about a verified user such as:
  - When the password is due to expire
  - When the user ID was last accessed
  - The date and time of the current verification
  - How many unauthorized attempts there have been for this user since the last valid access
- Specify a default userid and password to be used for communication over a CICS server connection.

To use the ESI interface, the CICS Universal Client or CICS Transaction Gateway must be connected to the CICS server with SNA. An ESM, such as Resource Access Control Facility (RACF), which is part of the z/OS Security Server, or an equivalent ESM, must also be available to the CICS server.

Chapter 5. Statistics API

This information describes the statistics API, which is provided only for the CICS Transaction Gateway.

If you want to use the statistics API in remote mode, you need to set up a statistics API protocol handler. See the *CICS Transaction Gateway Administration Guide* for your platform for information about how to do this.

Statistical data overview

The statistics API allows a single-threaded or multithreaded user application to access statistical data from one or more running Gateway daemons.

API functions

The API provides functions to:

- Connect to specific Gateway daemons, using gateway tokens.
- Disconnect from specific Gateway daemons, using gateway tokens.
- Obtain a set of statistical group IDs from a specific Gateway daemon.
- Obtain statistical IDs associated with one or more statistical group IDs from a specific Gateway daemon.
- Obtain data for statistical IDs from a particular Gateway daemon.

The functions are grouped into five categories:

- Connection functions
- ID data retrieval functions
- Statistical data retrieval functions
- Result set manipulation functions
- Utility functions

Calling the API

This section explains how applications invoke API functions.

Applications invoke API functions defined in "C-language header files" on page 29, and a dynamic link library (DLL). Each function call returns an integer result code, defined in the `ctgstdat.h` header file. A function that completes normally returns the code `CTG_STAT_OK`. A function that needs to report a problem returns a negative code, detailed in the `ctgstdat.h` header file.

The statistics API does not provide logging messages. Runtime operation of the API functions can be monitored using trace facilities. Statistics API tracing can be enabled programatically with data written to `stderr`, or a specified file. API errors are reported to the calling application using an integer result code.

API and protocol version control

A statistics API application, and the Gateway daemon providing the statistics, might be from different versions of the CICS Transaction Gateway.

API and protocol version control helps ensure that a statistics API application can issue meaningful requests to a CICS Transaction Gateway daemon, and get meaningful responses in return. API and protocol versions have a format of four digits, separated by the underscore character. For example: `1_0_0_0`

Note: The API and protocol versions might look like the product version, but they are not related. The statistics API can only be used to collect statistical data from Gateway daemons at version 7.0 or higher.

A statistics API application can:

- Find the protocol version that it was compiled with by using the compile-time string `CTG_STAT_PROTOCOL_VER`, defined in `ctgstdat.h`.
- Find the API version that it was compiled with by using the compile-time string `CTG_STAT_API_VERSION`, defined in `ctgstats.h`.
- Find which protocol version is used by a CICS Transaction Gateway daemon, by using the “openGatewayConnection” on page 37 or “openRemoteGatewayConnection” on page 37 function.
- Find which API version is provided by a CICS Transaction Gateway daemon, by using the “getStatsAPIVersion” on page 43 function.

The API version of the statistics API application must have the same major version number as the CICS Transaction Gateway daemon. If the major version numbers differ, API calls might fail. The minor version number of the CICS Transaction Gateway API version must be the same or greater than the API version of the statistics API application, otherwise some new API functions might not be available.

The protocol version of the statistics API application must have the same major version number as the CICS Transaction Gateway daemon. The minor protocol version number of the statistics API application must be the same or greater than the minor protocol version number of the CICS Transaction Gateway daemon. Otherwise, the statistics API application might be unable to interpret responses from function calls.

Statistics API components

The statistics API is made available to user applications by two C-language header files and a dynamic link library (DLL).

C-language header files

Two platform-independent C-language header files are provided for developing user applications.

`ctgstats.h` defines the API function calls and datatypes required to use the API functions.

`ctgstdat.h` defines the set of query return codes that might be seen by a statistical user application. The set of query return codes can vary according to the statistics protocol version provided by the CICS Transaction Gateway daemon.

Runtime DLL

The statistics API runtime DLL is provided for each of the supported CICS Transaction Gateway hardware platforms. It is supplied as a platform-specific DLL binary. It must be available during the runtime of the statistical user application.

Sample code

A sample file `ctgstat1.c` is supplied. This provides a simple example for using the statistics API. Further details of the `ctgstat1.c` sample are provided in the `samples.txt` file.

Runtime components for z/OS

This section describes the runtime components for z/OS.

Dataset names and SMP/E types

On z/OS, the runtime DLL and header file are delivered by SMP/E. The details are provided in Table 5.

Table 5. Dataset names and SMP/E types

Deliverable	**Distribution**	**Target**	**Member**	**Type**
DLL	*hlq*.ACTGMOD	*hlq*.SCTGDLL	CTGSTATS	++MOD
C Header	*hlq*.ACTGINCL	*hlq*.SCTGINCL	CTGSTATS	++SRC
C Header	*hlq*.ACTGINCL	*hlq*.SCTGINCL	CTGSTDAT	++SRC
C Sample	*hlq*.ACTGSAMP	*hlq*.SCTGSAMP	CTGSTAT1	++SRC

Table 5. Dataset names and SMP/E types (continued)

Deliverable	Distribution	Target	Member	Type
Sample JCL	*hlq*.ACTGSAMP	*hlq*.SCTGSAMP	CTGSTJOB	++SRC
Sidedeck	SMP/E *generated*	*hlq*.SCTGSID	CTGSTATS	Not applicable

The DLL load module is link-edited during installation. When the `SCTGDLL` library is added to the `STEPLIB` concatenation, user applications can use the statistics API. If the application uses implicit DLL loading, the sidedeck might be required to complete the link-edit cycle.

Runtime components for multiplatforms

This section describes the runtime components for multiplatforms.

File names and locations

The runtime DLL and header files are installed by the multiplatform installer. The details of the files are provided in Table 6.

Table 6. File names and locations

Platform	Deliverable	File name	Installation directory
All	C Header	`ctgstats.h`	`include`
All	C Header	`ctgstdat.h`	`include`
All	C Sample	`ctgstat1.c`	`samples/c/stats`
AIX®	DLL	`libctgstats.so`	`lib`
HP-UX	DLL	`libctgstats.sl`	`lib`
Linux on Intel®	DLL	`libctgstats.so`	`lib`
Linux on POWER™	DLL	`libctgstats.so`	`lib`
Linux on zSeries®	DLL	`libctgstats.so`	`lib`
Solaris	DLL	`libctgstats.so`	`lib`
UNIX and Linux	Sample Makefile	`samp.mak`	`samples/c/stats`
Windows	DLL	`ctgstats.dll`	`bin`
Windows	Export symbols	`ctgstats.lib`	`lib`
Windows	Sample Makefile	`ctgstat1mak.cmd`	`samples/c/stats`

For information on supported compilers, see the *CICS Transaction Gateway: Administration* book for your operating system.

Windows platform

At compile time, applications that use the statistics API need access to the API DLL external symbols provided in the `ctgstats.lib` file.

UNIX and Linux platforms

If you change the sample makefile, you might also have to update the `samples/c/env_c.def` file.

Statistics API program structure

Outline of a basic statistics API program.

A basic statistics API program typically has an outline similar to the example later in this section.

Example

This pseudo-code program connects to a CICS Transaction Gateway daemon, obtains the statistics IDs related to the ″GD″ resource group, obtains the current values for the given ″GD″ related statistical IDs and finally iterates through the returned values, writing out the details.

```
/* Create a connection to a local Gateway daemon */
openGatewayConnection(&gwyToken,port,&gwyProtocolVersPtr)

verify connected Gateway protocol level

/* Set the resource group id of interest */
queryString1="GD"

/* Obtain the list of associated statistical IDs */
getStatIdsByStatGroupId(gwyToken, queryString1, &resultSetToken)

/* Extract the returned IDs as a query string */
getIdQuery(resultSetToken,&queryString2)

/* Obtain the live statistical values for the given set IDs */
getStatsByStatId(gwyToken, queryString2, &resultSetToken)

/* Iterate over the result set, outputting */
/* the details of each result set element */

/* Obtain the first statistical result set element */
getFirstStat(resultSetToken, &statDataItem)

do
   if statDataItem.queryElementRC == CTGSTATS_SUCCESSFUL_QUERY
      /* output details of statDataItem */
   endif
   /* Obtain the next statistical result set element */
   getNextStat(resultSetToken, &statDataItem)
until end-of-resultset
```

API data types

Data types defined and used by the statistics API.

This information describes the main data types used by the statistics API.

Gateway tokens

A Gateway token represents a single connection to a specific Gateway daemon.

When a connection to a Gateway daemon is made, all subsequent API calls that retrieve statistical data must include the Gateway token as a parameter.

The statistics API handler in a Gateway daemon is restricted to five connection threads. This means that a single Gateway daemon can only deal with five connected statistics API programs, or threads, at the same time.

A statistical API program should avoid holding more than one connection to the same Gateway daemon at the same time.

A statistical API program can hold multiple Gateway tokens, but can only use them on the thread that called the “openGatewayConnection” on page 37 or “openRemoteGatewayConnection” on page 37 in order to retrieve the token.

A Gateway token type (`CTG_GatewayToken_t`) is defined in the “C-language header files” on page 29.

Query strings

A query string is an input parameter, specifying the statistical data to be retrieved.

A query string is an input parameter to statistical API functions which provide a result set token pointer. The string is a null-terminated, colon-separated list of IDs. The IDs can be statistical group IDs, or statistical IDs. An empty query string `""` is interpreted as matching all IDs appropriate to the function call.

Query strings are of type `(char *)`, and contain character data in the native encoding. The null terminator is added implicitly when creating strings in C using the `""` characters.

The user application creates and manages the query string character buffer.

Where an API function produces a data result set, the function “getIdQuery” on page 41 can be used to obtain a query string suitable for input to another API call.

Example

A pseudo-code example showing the query string used to retrieve the Gateway daemon status and all Connection Manager statistics is:

```
result = getStatsByStatId(gwyTok, "GD_CSTATUS:CM", &rsToken1);
```

Result set tokens

A result set token is a reference to a set of results from a single statistics API function call.

If a statistics API function calculates a set of data, the function provides a reference to the result set. This reference is called a result set token. The result set may contain either:

- ID data, including statistical group IDs or statistical IDs

or:

- Statistical data

A result set token is used to work with result set data. For example, a result set token enables a user application to browse through the result set, or extract specific details. The application can use functions such as “getFirstId” on page 41 or “getNextStat” on page 42 to manipulate the result set data.

An “ID data” on page 35 type is populated by the “getFirstId” on page 41 and “getNextId” on page 41 functions. A “Statistical data” on page 35 type is populated by the “getFirstStat” on page 42 and “getNextStat” on page 42 functions. The data types are used to access the data in the result sets, as described in “Correlating results and error checking” on page 45.

Note: All ID data and statistical data is in character format, using the default native string encoding.

Result set tokens returned by a statistics API function are ’owned’ by the API. The token is freed when either:

- The associated Gateway daemon connection is closed using the “closeGatewayConnection” on page 38 function.

or

- The function “closeAllGatewayConnections” on page 38 is called.

The result set token returned by the “copyResultSet” on page 42 function is *not* ’owned’ by the API. The token can only be freed using the “freeResultSet” on page 43 function.

Result set tokens ’owned’ by the API cannot be ’freed’ using the “freeResultSet” on page 43 function. The tokens must be freed using the “closeGatewayConnection” on page 38 or “closeAllGatewayConnections” on page 38 functions.

Result sets which are API-owned can only be manipulated on the thread which obtained them. Result sets that were not created by API calls can be manipulated by any thread.

Working with multiple result sets

Working with multiple result sets requires special attention.

Calling a statistics API function produces a result set token. This token identifies a result set owned by the statistics API. The result set is also associated with the Gateway identified by the gateway token used during the function call. This means that each result set owned by the statistics API is associated with a specific Gateway connection. It is helpful to think of the gateway token and the corresponding result set token as a pair.

Tokens referring to API-owned result sets may only be used by the thread which created them. To create a result set token usable by any thread, call the “copyResultSet” on page 42 function.

For example, an application using the same gateway token to make two separate API function calls will be given two logically different result set tokens. Since the same gateway token was used for both calls, the different result set tokens will iterate over the *same* result set. The result set will be the one returned by the last API function call.

This means that the result set identified by an result set token is only valid until another API call is made, specifying the same gateway token. The most recent API call overwrites the existing result set.

Use the “copyResultSet” on page 42 function to make a copy of a result set before it is overwritten by another API call. When the application finishes using the copied result set, free the storage using the “freeResultSet” on page 43 function.

Example

In the following example code, two statistics API calls are made. The same Gateway token is used for both calls. Two separate addresses are supplied for the result set tokens.

```
getStatsByStatGroupId(gwyTok, "", &rsTok1);
/* Tasks after getStatsByStatGroupId function call. */
getStatsByStatId(gwyTok, "", &rsTok2);
/* Tasks after getStatsByStatId function call. */
```

Using the same Gateway token both calls means that the result set pointed to by `&rsTok1` will be overwritten when the second API call is made. The two separate result set tokens `&rsTok1` and `&rsTok2` will iterate over the same result set.

If the result set obtained from the first API call is still required later in the application, take a copy of the result set by calling the “copyResultSet” on page 42 function.

ID data

An ID data structure maps an individual result returned from an ID API function.

The data type `CTG_IdData_t` is defined in the “C-language header files” on page 29. The data provides a name for individual results within statistical groups or statistics.

Individual results can be accessed using the “getFirstId” on page 41 and “getNextId” on page 41 functions.

`CTG_IdData_t` provides two fields, a character pointer and length, to enable access to individual elements of an ID result set, as described in “Correlating results and error checking” on page 45.

Statistical data

A statistical data structure maps an individual result returned from a statistics API function.

The data type `CTG_StatData_t` is defined in the “C-language header files” on page 29. The statistical data represents individual statistics, or name-value pairs.

Individual results can be accessed using the “getFirstStat” on page 42 and “getNextStat” on page 42 functions.

`CTG_StatData_t` provides two fields, a character pointer and length, to enable access to individual elements of a statistical result set. These elements are the statistical ID and statistical value data, as described in “Correlating results and error checking” on page 45.

Statistics API Trace Levels

The CICS Transaction Gateway statistics API provides several levels of diagnostic trace information.

Trace Levels

The CICS Transaction Gateway statistics API can produce diagnostic trace information, depending on the trace level setting.

Each level automatically includes all the detail provided by the lower levels. For example, `CTG_STAT_TRACE_LEVEL2` indicates that all events and exceptions will be traced.

Table 7. Statistics API Trace Levels

Trace level	**Output details**
`CTG_STAT_TRACE_LEVEL0`	No trace output.
`CTG_STAT_TRACE_LEVEL1`	Exceptions only.
`CTG_STAT_TRACE_LEVEL2`	Events.
`CTG_STAT_TRACE_LEVEL3`	Entries and exits.
`CTG_STAT_TRACE_LEVEL4`	Debug information.

The default trace destination is `stderr`. Use the function “setAPITraceFile” on page 44 to choose a different trace destination.

API functions

The statistics API functions.

This information describes the functions provided in the statistics API.

Many ID functions create a result set. A result set is over-written the next time an ID function call is made using the same gateway token. This means an application working with several result sets from the same Gateway connection at the same time must take a local copy of each result set. To take a local copy of a result set, use the “copyResultSet” on page 42 function.

For details of the return codes provided by the API functions, see `ctgstats.h` in the “C-language header files” on page 29, or refer to the *CICS Transaction Gateway: Programming Reference*.

Gateway daemon connection functions

This information describes the main functions provided in the statistics API for connections to a Gateway daemon.

openGatewayConnection

This function establishes a connection to a local Gateway daemon statistics protocol handler, using the specified port number, a pointer to a gateway token, and the address of a `char` pointer for the statistics API protocol version.

Detail

This function is called with an integer for the target port number, a pointer to a gateway token, and the address of a `char` pointer to hold a string describing the version of the statistics API protocol provided by the target gateway daemon.

The function creates a connection to a local Gateway daemon statistics protocol handler using the specified port number.

When the call returns, the gateway token represents the connection to the specified Gateway daemon. The token is required to interact with that Gateway daemon in subsequent API calls.

The `char` pointer points to a null-terminated character string. The API owns the storage for the protocol version character array, and the API program should not free this storage.

The user application must check that the version of the statistics API protocol provided by the target Gateway daemon is at least the same as major version number in the compile-time string `CTG_STAT_PROTOCOL_VER`. This compile-time string is defined in `ctgstdat.h`, described in the "C-language header files" on page 29 section. The major version number is the first digit in the compile-time string.

openRemoteGatewayConnection

This function establishes a connection to a remote Gateway daemon statistics protocol handler, using the specified host name, port number, a pointer to a gateway token, and the address of a `char` pointer for the statistics API protocol version.

Detail

This function is called with:

- A character pointer for the hostname. This is a null terminated string containing the IP address or hostname of the machine running the Gateway daemon.
- An integer for the target port number.
- A pointer to a gateway token.

- The address of a `char` pointer to hold a string describing the version of the statistics API protocol provided by the target gateway daemon.

The function creates a connection to a remote Gateway daemon statistics protocol handler using the specified port number.

When the call returns, the gateway token represents the connection to the specified Gateway daemon. The token is required to interact with that Gateway daemon in subsequent API calls.

The `char` pointer points to a null-terminated character string. The API owns the storage for the protocol version character array, and the API program should not free this storage.

The user application must check that the version of the statistics API protocol provided by the target Gateway daemon is at least the same as major version number in the compile-time string `CTG_STAT_PROTOCOL_VER`. This compile-time string is defined in `ctgstdat.h`, described in the "C-language header files" on page 29 section. The major version number is the first digit in the compile-time string.

closeGatewayConnection

This function closes a connection to a Gateway daemon statistics protocol handler, using the gateway token provided.

Detail

This function is called with a pointer to a gateway token. The function closes the connection to the local or remote Gateway daemon statistics protocol handler identified by the gateway token. Any resources associated with the connection, including result sets, are freed, and result set tokens obtained with the specified gateway token are no longer valid.

When the call returns, the gateway token pointer is set to null, showing that it is no longer valid.

closeAllGatewayConnections

This function releases all resources owned by the statistics API, including any open Gateway daemon connections.

Detail

An application can use this function as part of a normal shutdown. The function can also be used in the event of a severe error, for example where some form of controlled shutdown is required but references to gateway tokens have been lost.

Copied result sets are not be freed by this function, because the API does not own or maintain a record of copied result sets.

ID functions

This information describes the ID functions provided in the statistics API.

getResourceGroupIds

This function returns a result set token, representing the set of resource group IDs currently available for the specified Gateway daemon.

Detail

This function is called with a gateway token and a result set token pointer. The result set returned can be parsed with functions "getFirstId" on page 41 and "getNextId" on page 41, or used to generate a query string with "getIdQuery" on page 41.

For the z/OS platform, depending upon when "getResourceGroupIds" is called, dynamic resource groups for a specific CICS server might not be returned in the list. The dynamic list of server resource group IDs can be obtained directly via the appropriate resource group statistical ID.

getStatIds

This function returns a result set token, representing the set of all statistical IDs currently available for the specified Gateway daemon.

Detail

This function is called with a gateway token and a result set token pointer. The result set created may be parsed with functions "getFirstId" on page 41 and "getNextId" on page 41, or used to generate a query string with "getIdQuery" on page 41.

getStatIdsByStatGroupId

This function returns a set of statistical IDs associated with the statistical group IDs supplied in the query string, for the specified Gateway daemon.

Detail

This function is called with a gateway token, a query string of statistical group IDs, and a result set token pointer. The result set created may be parsed with functions “getFirstId” on page 41 and “getNextId” on page 41, or used to generate a query string with “getIdQuery” on page 41.

Retrieving statistical data functions

This information describes the data retrieval functions provided in the statistics API.

getStats

This function creates a result set token representing the set of all available statistical name-value pairs for the specified Gateway daemon.

Detail

This function is called with a gateway token and a result set token pointer. The result set created may be parsed with functions “getFirstStat” on page 42 and “getNextStat” on page 42, or used to generate a query string with “getIdQuery” on page 41.

getStatsByStatId

This function creates a result set token. The token represents the set of name-value pairs that is generated when a query string of statistical IDs is applied to the specified Gateway daemon.

Detail

This function is called with a gateway token, a query string of statistical IDs, and a result set token pointer. The result set created can be parsed with functions “getFirstId” on page 41 and “getNextId” on page 41, or used to generate a query string with “getIdQuery” on page 41.

getStatsByStatGroupId

This function creates a result set token. The token represents the set of name-value pairs that is generated when a query string containing statistical group IDs is applied to the specified Gateway daemon.

Detail

This function is called with a gateway token, a query string of statistical group IDs, and a result set token pointer. The result set returned can be parsed with functions “getFirstStat” on page 42 and “getNextStat” on page 42, or used to generate a query string with “getIdQuery” on page 41.

Result set functions

This information describes the result set functions provided in the statistics API.

getIdQuery

This function provides a pointer to a character array, containing the ID result set.

Detail

This function is called with a result set token pointer, and the address of a character pointer. The function sets the pointer to point to a character array. This character array contains the ID result set, formatted for direct use as a query string.

The storage for the character array is created by the API. The API owns the storage for the character array, and the API program should not free this storage.

getFirstId

This function populates a `CTG_IdData_t` variable with details of the first ID in a result set.

Detail

This function is called with an ID result set token. The function populates a `CTG_IdData_t` variable with details of the first ID in the result set. If there are no further IDs in the result set, the `CTG_IdData_t` variable is unchanged.

For more information on the `CTG_IdData_t` data type, see "ID data" on page 35

getNextId

This function populates a `CTG_IdData_t` variable with details of the next ID in a result set.

Detail

This function is called with an ID result set token. The function populates a `CTG_IdData_t` variable with details of the next ID in the result set. If there are no further IDs in the result set, the `CTG_IdData_t` variable is unchanged.

For more information on the `CTG_IdData_t` data type, see "ID data" on page 35

getFirstStat

This function populates a `CTG_StatData_t` variable with details of the first ID in a result set.

Detail

This function is called with a statistical result set token. The function populates a `CTG_StatData_t` variable with details of the first ID in the result set. If there are no further IDs in the result set, the `CTG_StatData_t` variable is unchanged.

For more information on the `CTG_StatData_t` data type, see "Statistical data" on page 35.

getNextStat

This function populates a `CTG_StatData_t` variable with details of the next ID in a result set.

Detail

This function is called with a statistical result set token. The function populates a `CTG_StatData_t` variable with details of the next ID in the result set. If there are no further IDs in the result set, the `CTG_StatData_t` variable is unchanged.

For more information on the `CTG_StatData_t` data type, see "Statistical data" on page 35.

copyResultSet

This function creates a copy of a result set. The copy is owned by the calling application.

Detail

An application might need to make several API calls on a result set. This is useful because some API calls overwrite an existing result set with new results. A local copy of the result set is created using this function.

The `copyResultSet` function takes two result set tokens. The source token refers to the original result set. The target token refers to a copy of the result set. The copy is created by this function. The calling application owns the target result set.

There is no structural difference between the original and the target result sets. "Result set functions" on page 41 work with API-owned result sets or application-owned result sets.

When the application finishes using the copied result set, free the storage using the “freeResultSet” function.

freeResultSet

This function frees the storage used by an application-owned result set.

Detail

When an application finishes using a result set, the storage must be freed. This function takes a pointer to a result set token, frees the storage, and sets the pointer to null.

This function should only be used for result sets created using the “copyResultSet” on page 42 function. If the result set is owned by the statistics API, an attempt to free the result set returns an error.

Utility functions

This information describes the utility functions provided in the statistics API.

getStatsAPIVersion

This function provides version information about the statistics API.

Detail

This function takes the address of a character pointer to be modified. The function modifies the character pointer to point to a null-terminated character array. The string represents the version of the active statistics DLL. Version information is described in “API and protocol version control” on page 28. The API owns the storage for the character array, and the API program should not free this storage.

getAPITraceLevel

This function provides information about the current trace status of the statistics API.

Detail

This function takes a pointer to a local `int` variable. The function sets the variable to the current trace level of the statistics API.

The levels are defined in the “C-language header files” on page 29. Valid values are:

- `CTG_STAT_TRACE_LEVEL0`
- `CTG_STAT_TRACE_LEVEL1`
- `CTG_STAT_TRACE_LEVEL2`

- `CTG_STAT_TRACE_LEVEL3`
- `CTG_STAT_TRACE_LEVEL4`

For further information on trace levels, see "Statistics API Trace Levels" on page 36.

setAPITraceLevel

This function sets the trace level of the statistics API.

Detail

This function takes an `int` value. The function sets the trace level of the API to this value.

The default trace destination is `stderr`. Use the function "setAPITraceFile" to choose a different trace destination.

The status values are defined in the "C-language header files" on page 29. Valid values are:
- `CTG_STAT_TRACE_LEVEL0`
- `CTG_STAT_TRACE_LEVEL1`
- `CTG_STAT_TRACE_LEVEL2`
- `CTG_STAT_TRACE_LEVEL3`
- `CTG_STAT_TRACE_LEVEL4`

For further information on trace levels, see "Statistics API Trace Levels" on page 36.

setAPITraceFile

This function sets the destination for statistics API trace details.

Detail

This function takes a character pointer to a null-terminated string. The string is the file name of the intended trace destination.

If the file name already exists, trace data is appended to the file.

If the file name cannot be opened for writing, trace data is sent to `stderr`.

Passing a null pointer to this function sets the trace destination back to `stderr`.

dumpResultSet

This function outputs a result set in a printable form.

Detail

This function takes a result set token. The function writes the contents of the result set to the trace destination, regardless of the current trace level. The contents are written using printable characters.

This function is normally used for debug purposes.

Related reference

"Statistics API Trace Levels" on page 36
The CICS Transaction Gateway statistics API provides several levels of diagnostic trace information.

dumpState

This function outputs internal information about the API.

Detail

This function writes internal information about the API to the trace destination.

This function is normally used for debug purposes.

Correlating results and error checking

Individual results within a result set from a statistics API function call can be correlated back to the original query string data.

ID or statistical results within a result set from an API call can be correlated back to the original query string data using the `struct` elements `queryElementPtr` and `queryElementLen`. The status of the result is given by `queryElementRC`. These return codes are defined in the `ctgstdat.h` header file.

After a call to "getFirstId" on page 41 or "getNextId" on page 41, the `CTG_IdData_t` elements `query` and `queryLen` represent the specific ID in the query string associated with the result.

After a call to "getFirstStat" on page 42 or "getNextStat" on page 42, the `CTG_StatData_t` elements `query` and `queryLen` represent the specific statistic in the query string associated with the result.

If the specific ID in the query string is in error, the `struct` element `queryElementRC` will have a non-zero value, defined in the `ctgstdat.h` header file.

Chapter 6. Programming in Java

This information provides an introduction to writing Java Client programs for the CICS Transaction Gateway.

Overview of the programming interface for Java

The CICS Transaction Gateway enables Java Client applications to communicate with programs on a CICS server by providing base classes for the External Call Interface (ECI) and the External Security Interface (ESI), and EPI support classes for the External Presentation Interface (EPI). The classes listed below are the basic classes provided with the CICS Transaction Gateway. For a full description of all the classes and methods referred to in this chapter, refer to the Javadoc supplied with the CICS Transaction Gateway.

Note that the EPI classes are not available with the CICS Transaction Gateway for z/OS.

com.ibm.ctg.client.JavaGateway
This class is the logical connection between a program and a CICS Transaction Gateway. You need a JavaGateway object for each CICS Transaction Gateway that you want to talk to.

com.ibm.ctg.client.ECIRequest
This class contains the details of an ECI request to the CICS Transaction Gateway.

com.ibm.ctg.epi.Terminal
This class controls a 3270 terminal connection to CICS. The Terminal class handles CICS conversational, pseudoconversational, and ATI transactions. A single application can create many Terminal objects.

com.ibm.ctg.client.ESIRequest
This class contains the details of an ESI request to the CICS Transaction Gateway.

Note: The **com.ibm.ctg.client.EPIRequest** base class is supported only for compatibility with earlier releases of the product. New programs should use the EPI support classes.

Creating a JavaGateway

The JavaGateway object is a logical connection between your application and the Gateway daemon when the application is running in remote mode. If a

Java Client application is running in local mode, the JavaGateway is a connection between the application and the CICS server, bypassing the Gateway daemon.

Use one of the constructors provided to create a JavaGateway. You must specify the protocol you are using, and the network address and port number of the remote Gateway daemon. You can specify this information either by using the setAddress, setProtocol and setPort methods, of the JavaGateway class, or by providing all the information in URL form: **Protocol://Address:Port**. If you specify a local connection, you must specify a URL of **local:** You can use the setURL method or pass the URL into one of the JavaGateway constructors.

Note: The IP address can be in IPv6 format. If you are using a Java Client application on an HP-UX system, and the application calls a Gateway that binds to an IPv6 address, specify `-Djava.net.preferIPv4Stack=false` explicitly.

The JavaGateway supports the following protocols :

- TCP/IP
- SSL
- Local

There are several constructors available for creating a JavaGateway. The default constructor creates a JavaGateway with no properties. You must then use the set methods to set the required properties and the open method to open the Gateway. There are other constructors which set different combinations of properties and open the Gateway for you.

Java Client application suite select feature

Cipher suites define the key exchange, data encryption, and hash algorithms used for an SSL session between a client and server. During the SSL handshake, both sides present the cipher suites that they are able to support and the strongest one common to both sides is selected. In this way, you can restrict the cipher suites that a Java Client application presents.

Restricting cipher suites for a Java Client application

To restrict the cipher suites used by a JavaGateway object, use the setProtocolProperties() method to add the property (JavaGateway.SSL_PROP_CIPHER_SUITES) to the properties object passed to it. The value of the property must contain a comma-separated list of the cipher suites that the application is restricted to using.

For example:

```
Properties sslProps = new Properties();
sslProps.setProperty(JavaGateway.SSL_PROP_KEYRING_CLASS, strSSLKeyring);
sslProps.setProperty(JavaGateway.SSL_PROP_KEYRING_PW, strSSLPassword);
sslProps.setProperty(JavaGateway.SSL_PROP_CIPHER_SUITES,
                     "SSL_RSA_WITH_NULL_SHA");
javaGatewayObject = new JavaGateway(strUrl, iPort, sslProps);
```

Writing Java Client applications

Before a Java Client application can send a request to the CICS server, it must create and open a JavaGateway object.

When the JavaGateway is open, the Java Client application can flow requests to the CICS server using the flow method of the JavaGateway. The request is sent to the Gateway daemon if you have a remote JavaGateway or direct to CICS if you are using a local JavaGateway on z/OS or direct to the Client daemon if you are using a local JavaGateway on multiplatforms.

When there are no more requests for the CICS Transaction Gateway, the Java Client application closes the JavaGateway object.

Deploying remote Java Client applications

You are licensed to copy the following files to the computer that is running the Java Client application:

Non-J2EE applications
File ctgclient.jar

J2EE applications in a managed environment
The resource adapters (RAR files) in the <install_path>\deployable directory.

J2EE applications in a non-managed environment
The following files in the <install_path>\classes directory:

cicsj2ee.jar
ctgclient.jar
ccf2.jar
connector.jar
screenable.jar

Ensure that any JAR files that you copy are listed on the class path of the remote computer.

JavaGateway security

When you connect to a remote CICS Transaction Gateway, resources allocated to a particular connection, and identifiers specified on the request objects on a particular connection, are available only to that connection. If you specify the

local: protocol, all JavaGateways that are created in the same JVM, that is, the same process, have access to each other's allocated resources or specified identifiers.

Making External Call Interface calls from a Java Client program

This section describes how to run a program on a CICS server using ECI calls from a Java Client application. Use the com.ibm.ctg.client.ECIRequest base class and the JavaGateway flow method to pass details of an ECI request to the CICS Transaction Gateway. Table 8 shows Java objects corresponding to the ECI terms described in "Input and output information for external calls to CICS" on page 9.

Table 8. ECI terms and corresponding Java objects

ECI term	Java object.field or object.method()
Abend code	ECIRequest.Abend_Code
Channel	setChannel(Channel)
COMMAREA	ECIRequest.Commarea
ECI timeout	ECIRequest.setECITimeout(short)
LUW control	ECIRequest.extend_mode
LUW identifier	ECIRequest.Luw_Token
Password	ECIRequest.Password
Program name	ECIRequest.Program
Server name	ECIRequest.Server
SocketConnectTimeout	ECIRequest:SocketConnectTimeout
TPNName	ECIRequest.Call_Type = ECI_SYNC_TPN or ECI_ASYNC_TPN
TranName	ECIRequest.Call_Type = ECI_SYNC or ECI_ASYNC
User ID	ECIRequest.Userid

Linking to a CICS server program

Use one of the ECIRequest constructors provided to set the required parameters for the ECI call. You can either use the default constructor which sets all parameters to their default values, or one of the other constructors which allow you to set different combinations of parameters. Place any data to be passed to the server program in a COMMAREA or container.

You can create ECI requests for synchronous and asynchronous program link calls by setting the value of Call_Type to ECI_SYNC or ECI_ASYNC.

If you use the ECI_ASYNC call type with CICS Transaction Gateway for z/OS, you must use the Callbackable interface.

Creating channels and containers for ECI calls

You can use channels and containers when you use connect to CICS using the IPIC protocol. You must construct a channel and add containers to the channel before it can be used in an ECIRequest.

1. Add the following code to your application program, to construct a channel to hold the containers:

   ```
   Channel myChannel = new Channel("CHANNELNAME")
   ```

2. You can add containers with a data type of BIT or CHAR to your channel. Here is a sample BIT container:

   ```
   byte[] custNumber = new byte[]{0,1,2,3,4,5};
   myChannel.createContainer("CUSTNO", custNumber);
   ```

   and a sample CHAR container:

   ```
   String company = "IBM";
   myChannel.createContainer("COMPANY", company);
   ```

3. The channel and containers can now be used in an ECIRequest, as the example shows:

   ```
   ECIRequest eciReq = new ECIRequest("CICSA", "USERNAME", "PASSWORD",
           "CHANPROG",channel, ECIRequest.ECI_NO_EXTEND, 0); eciReq.flow();
   ```

4. When the request is complete, you can retrieve the current state of the containers in the channel, as the example shows:

   ```
   Channel myChannel = eciReq.getChannel();

   for(Container container: myChannel.getContainers()){
      System.out.println(container.getName());

      if (container.getType() == ContainerType.BIT){
         byte[] data = container.getBITData();
      }
      if (container.getType() == ContainerType.CHAR){
         String data = container.getCHARData();
      }
   }
   ```

   If you are using this channel in an extended request, you must use the same channel object in subsequent flows.

Managing a LUW

Set the call type to ECI_EXTEND if the ECI call is part of an extended LUW. If the call is the last, or only call for the LUW, the call type must be ECI_NOEXTEND, ECI_COMMIT or ECI_BACKOUT.

Retrieving replies from asynchronous requests

Callbacks

ECIRequest supports callback objects. A callback object, which must implement the **Callbackable** interface, receives the results of the flow via the setResults method. When the results have been applied, a new thread is started to execute the run method.

If you specify a callback object for a synchronous call the results are passed to your Callbackable object as well as to your ECIRequest object in the flow request.

Reply solicitation calls

Use the **automatic message qualifier generator** feature of ECIRequest to ensure that the message qualifiers that you assign are unique within the CICS Transaction Gateway. Turn the feature on by invoking the method setAutoMsgQual(true) on your ECIRequest object. This will assign a message qualifier that is unique on all asynchronous requests (ECI_ASYNC, ECI_ASYNC_TPN, ECI_STATE_ASYNC, ECI_STATE_ASYNC_JAVA), when the request is flowed. Use this message qualifier to retrieve replies when you use the ECI_GET_SPECIFIC_REPLY and ECI_GET_SPECIFIC_REPLY_WAIT call types.

For remote connections you cannot get replies on a different connection to the one that flowed the original request with a message qualifier; see "ECI security" on page 54.

If you use ASYNC calls with message qualifiers, you might have to pass a user ID and password when you retrieve the reply with one of the various GET_REPLY call types. The user ID and password are not used to validate whether the reply can be retrieved; they are passed to the Gateway to hold in case security is required to clean up (BACKOUT) an LUW if the connection is lost while the server program is still running.

For a local connection, the message qualifier should be unique for each request, although this is not enforced. Provided the JavaGateways are within the same JVM, any connection can get a message using a message qualifier, even if the request was flowed over a different connection. However, it is recommended that you use automatic message qualifier generation:

- To avoid problems resulting from reusing the same message qualifier
- To allow you to switch your application between local and remote connection

If you are using the CICS Transaction Gateway for z/OS, the following considerations apply:

- You cannot use the variable Message_Qualifier, or the methods isAutoMsgQual(), setAutoMsgQual(), setMessageQualifier(), or setMessageQualifier()
- You cannot use reply solicitation call types such as GET_REPLY, GET_REPLY_WAIT, GET_SPECIFIC_REPLY, or GET_SPECIFIC_REPLY_WAIT

ECI timeouts

If you are not using z/OS, use the ECIRequest.setECITimeout() method to set the value of ECI timeout.

If you are using z/OS, you cannot use the methods getECITimeout(), or setECITimeout(). You have the following options available, depending on whether you are using EXCI or IPIC:

- If you are using EXCI, you can set the TIMEOUT parameter in the EXCI options table DFHXCOPT.
- If you are using IPIC in remote mode, you can set the CONNECTTIMEOUT parameter in the configuration file.
- If you are using IPIC in local mode, you can set this in the JavaGateway.setSocketConnectTimeout() method.

For more information on ECI timeouts, see the *CICS External Interfaces Guide*, SC34-6006.

See "Timeout of the ECI request" on page 14

Performance considerations when transmitting data in a COMMAREA

The length of data transmitted between the user application and the server program can be much smaller than the size of the COMMAREA. You can use one of the methods listed below to prevent null values or other unwanted data from being transmitted over the network:

- The **setCommareaOutboundLength** method ensures that you send only the required data in the outbound flow to CICS, not the full Commarea_Length.
- The **getInboundDataLength** method shows the amount of non-null data returned.
- The **setCommareaInboundLength** method can be used if the unwanted data in the COMMAREA contains something other than nulls.

In order to reduce network traffic, the CICS Transaction Gateway truncates COMMAREAs by removing trailing null values. This process is called null stripping.

You can use the setCommareaOutboundLength method to ensure that only the data required by the server program is transmitted.

You can use the setCommareaInboundLength method to restrict the length of the data that the Java Client application receives back from CICS. If you use setCommareaInboundLength, the CICS Transaction Gateway does not remove trailing nulls from the inbound data.

The CICS server adds trailing nulls to the data received to extend it to the length specified in Commarea_Length so that the server program always receives a full COMMAREA.

The server program only returns the data required by the Java Client application. The Client daemon adds trailing nulls to the data received so that the CICS Transaction Gateway receives a full COMMAREA.

The CICS Transaction Gateway strips the trailing nulls from the COMMAREA before transmitting the data to the Java Client application.

The Java Client application must add nulls to the data to bring it up to the full size of the COMMAREA.

If the Java Client application is running locally on z/OS, a similar procedure takes place, but the improvement in performance is not as great.

ECI security

Logical unit of work (LUW) IDs and message qualifiers can only be used on the JavaGateway that created or assigned them. This is a security feature. It stops programs that are connected to the same CICS Transaction Gateway from using LUW IDs belonging to another application, or from using its message qualifier to request messages. For example, attempts to get a specific reply to a message from a different JavaGateway will result in an ECI_ERR_NO_REPLY return code.

EXCI security for ECI requests on z/OS

If you are using the EXCI protocol on CICS Transaction Gateway for z/OS, there are a number of settings and security checks in place to validate your user ID and password.

- The user ID and password coded on the ECI request object can be validated in the CICS Transaction Gateway through RACF for every EXCI call. This is controlled through the setting of the AUTH_USERID_PASSWORD environment variable. See the *CICS Transaction Gateway: z/OS Administration*, for more information.
- The ECI user ID will then be subject to EXCI surrogate security checks, before it can be flowed on the EXCI request; for more details, see the *CICS External Interfaces Guide*. Note that any password supplied on an ECI request is not flowed on to CICS from the CICS Transaction Gateway for z/OS.

- The flowed user ID is subject to CICS authorization checks, for more details, see theCICS Transaction Server for z/OS RACF Security Guide.

See also Configuring CICS Transaction Gateway for use with RACF, in the CICS Transaction Gateway: z/OS Administration.

IPIC security

IPIC connections enforce link security to control user activity over a connection, and flowed security to allow you to specify a username and password before communicating with a secured CICS region.

To set up user security, you need to define an IPCONN definition in CICS, that relates to the APPLID defined by theCICS Transaction Gateway or resource adapter.

The USERAUTH setting in the IPICONN definition is comparable to the ATTACHSEC setting. USERAUTH=IDENTIFY allows only SSL client authentication and communication between programs within a sysplex. All other communications require USERAUTH=VERIFY.

IPIC link security

There are two ways that you can specify the link user for IPIC connections. You can use the SECURITYNAME option, or an SSL certificate. You can use an SSL certificate if you have a client authenticated SSL (this is where both the client and server have certificates). The client's certificate is mapped by RACF to a specific user ID, which is defined as the link user. This means that you can specify different link users, depending on which certificate you are using.

To specify a link user, you must do the following:

1. Define an IPCONN definition in CICS, that relates to the APPLID defined by theCICS Transaction Gateway or resource adapter.
2. Set LINKAUTH to either:
    a. SECUSER if you want to use SECURITYNAME
    b. CERTUSER if you want to use the SSL certificate
3. If you specify SECUSER, specify the SECURITYNAME option.
4. If you specify CERTUSER, define your mappings in RACF to your chosen user ID. Ensure you are using a TCPIPSERVICE definition that is set up for SSL (not TCP) and is also enabled for client authentication.

When you specify CERTUSER, RACF maps the client certificate to a user ID. CICS defines this user ID as the link user. This process is called Certificate name filtering. For more information on Certificate name filtering, see the IBM Redpaper *J2C Security on z/OS* (redp4202.pdf) at the IBM Redbooks® Web site.

IPIC flowed security

You can specify a user ID and password before setting up a connection to a secured CICS region, either by using the ECIRequest base class, or by setting variables on the object.

To set custom properties for the ECI resource adapter:

- Set the flowed username in the UserName property
- Set the password in the Password property

To override ECIConnectionSpec settings:

- Create an ECIConnectionSpec object with the required username and password.
- Use this object for requests on the selected connection and in the getConnection() method of your ECI ConnectionFactory.

ECI return codes and server errors on z/OS

This section describes how the return codes from the EXCI are returned to the user of the **ECIRequest** object.

Table 9 shows how EXCI return codes map to ECI return codes. The EXCI return codes are documented in the *CICS External Interfaces Guide*.

Table 9. EXCI return codes and ECI return codes

EXCI return codes	ECI symbolic names/return codes	rc
201, 203	ECI_ERR_NO_CICS	–3
202	ECI_ERR_RESOURCE_SHORTAGE	–16
401, 402, 403, 404, 410, 411, 412, 413, 418, 419, 421	ECI_ERR_SYSTEM_ERROR	–9
422	ECI_ERR_TRANSACTION_ABEND	–7
423	ECI_ERR_SECURITY_ERROR	–27
601, 602, 603, 604, 605, 606, 607, 608, 621, 622, 623, 627, 628	ECI_ERR_SYSTEM_ERROR	–9
609	ECI_ERR_SECURITY_ERROR	–27
624	ECI_ERR_REQUEST_TIMEOUT	–5

EXCI support on z/OS

Version 2 of the EXCI is supported, and it provides support for **eci_transid** and resolves previous problems with ASCII/EBCDIC conversion.

If you use EXCI Version 2 and **eci_tpn** is specified on the ECI request, then the definition of the user mirror transaction must now specify PROGRAM(DFHMIRS), regardless of whether the transaction is defined as local or remote.

IPIC support for ECI

IPIC connections do not support ECI State calls or asynchronous ECI calls that use message qualifiers. If you are using local mode, IPIC connections are not displayed in the CICS_ECIListSystems call.

IPIC does not support the following ECI calls:
- ECI_ASYNC, with a message qualifier (Callbackable objects are supported)
- ECI_ASYNC_TPN, with a message qualifier (Callbackable objects are supported)
- ECI_GET_REPLY
- ECI_GET_REPLY_WAIT
- ECI_GET_SPECIFIC_REPLY
- ECI_GET_SPECIFIC_REPLY_WAIT
- ECI_STATE_ASYNC
- ECI_STATE_ASYNC_JAVA
- ECI_STATE_CANCEL
- ECI_STATE_CHANGED
- ECI_STATE_IMMEDIATE
- ECI_STATE_SYNC
- ECI_STATE_SYNC_JAVA

If you are using local mode, IPIC servers are not displayed in a CICS_EciListSystems call. This is because the IPIC information is passed using a URL and is not known in advance of the connection. However, if you are using remote mode, you define your IPIC servers in the configuration file (the URL function is not available for remote mode), and the servers are displayed in the CICS_EciListSystems call.

Making External Presentation Interface Calls from a Java Client Program

This section describes how to run a 3270–based program on a CICS server using EPI calls from a Java Client application. To do this you can use either the EPI support classes, which is the recommended method, or the EPIRequest base class. Neither of these methods is available for the CICS Transaction Gateway for z/OS. Table 10 on page 58 shows Java objects corresponding to the EPI terms described in “Terminal characteristics” on page 19.

Table 10. EPI terms and corresponding Java objects

EPI term	Terminal object:property	EpiRequest object.field
Code page	Terminal:CCSid	EPIRequest.CCSid
Color	no equivalent	EPIRequest.color
Columns	Screen:Width	EPIRequest.numColumns
Device type	Terminal:Device type	EPIRequest.deviceType
Error last line	no equivalent	EPIRequest.errLastLine
Error message color	no equivalent	EPIRequest.errColor
Error message highlight	no equivalent	EPIRequest.errHighlight
Error message intensity	no equivalent	EPIRequest.errIntensity
Extended highlight	no equivalent	EPIRequest.highlight
Install timeout	Terminal:InstallTimeout	EPIRequest.installTimeout
Map name	Screen:MapName	EPIRequest.mapName
Mapset name	Screen:MapsetName	EPIRequest.mapSetName
Maximum data	no equivalent	EPIRequest.maxData
Netname	Terminal:Netname	EPIRequest.netName
Password	Terminal:Password	EPIRequest.password
Read timeout	Terminal:ReadTimeout	EPIRequest.readTimeout
Rows	Screen:Depth	EPIRequest.numLines
Server name	Terminal:ServerName	EPIRequest.Server
Sign-on capability	Terminal:SignonCapability	EPIRequest.signoncapability
SocketConnectTimeout	No equivalent	EPIRequest:SocketConnectTimeout
Terminal ID	Terminal:Termid	EPIRequest.termID
Userid	Terminal:Userid	EPIRequest.userid

EPI support classes

This section:

- Explains how to use the EPI support classes
- Describes how to handle exceptions
- Describes the encoding of 3270 data streams
- Explains how to convert BMS maps and use the Map class
- Describes how to use the EPIRequest class

The CICS Transaction Gateway EPI support classes make it simpler for a Java programmer to access the facilities that the EPI provides:

- Adding and deleting terminals

- Starting CICS transactions
- Sending and receiving 3270 data streams

You do not need a detailed knowledge of 3270 data streams. EPI support classes provide higher-level constructs for handling 3270 data streams:

- General purpose Java classes are provided for handling screens, terminal attributes, and transaction data.
- Java classes for specific CICS applications can be generated from BMS map source files. These classes allow Java Client applications to access data on 3270 panels, using the same map field names used in the CICS program.

Note: These classes do not contain any specific support for 3270 data streams that contain DBCS fields. Data streams with a mixture of DBCS and SBCS fields are not supported.

The BMS conversion utility is a tool for statically producing Java class source code from a CICS BMS map set. See "Converting BMS maps and using the Map class" on page 68.

The EPI support classes are similar to the C++ EPI classes in that the objects required and the methods to manipulate them are similar.

In the examples in this chapter, statements similar to the following are assumed:

```
import com.ibm.ctg.epi.*; import java.io.*;
```

Adding a terminal to CICS

This section describes how to install a terminal on a CICS server. For more information about EPI and terminal properties, such as Sign-on capability and Read timeout, see Chapter 3, "External Presentation Interface (EPI)," on page 17.

EPIGateway: Create a JavaGateway object to start a connection to the CICS Transaction Gateway before attempting to connect a terminal to CICS. The EPIGateway class provides methods to access information about CICS servers that are accessible from the CICS Transaction Gateway, and it can be used instead of the JavaGateway class.

Adding a basic terminal: There are two ways to construct a basic terminal:

- Using the default constructor
- Using the basic terminal constructor

Default terminal constructor

To create a terminal using the default constructor, first instantiate a terminal, and then use the appropriate setter methods to set the required properties. Use only the setters that apply to a basic terminal. These methods are:

- setGateway
- setServerName
- setDeviceType
- setNetName
- setSession

All the set methods, with the exception of setGateway, are optional and have a default setting of null. After you have defined your terminal, install it on the CICS server using the connect() method. Use only this version of the connect() method. The connect(installTimeout) and connect(Session, InstallTimeout) methods are allowed only for extended terminals. See "Installing a terminal on CICS" on page 62 and "Synchronization and sessions" on page 64 for further information.

```
try {
    EPIGateway eGate = new EPIGateway("tcp://MyGateway",2006);
    Terminal term = new Terminal();
    term.setGateway(eGate);
    term.setServerName("CICS1");
    term.connect();
}
catch (IOException ioEx) {
    ioEx.printStackTrace();
}
catch (EPIException epiEx) {
    epiEx.printStackTrace();
}
```

Basic terminal constructor

The second way is to use the basic terminal constructor. This sets all the required properties and automatically connects you to the CICS Server.

```
try {
    EPIGateway eGate = new EPIGateway("tcp://MyGateway",2006);
    Terminal term = new Terminal(eGate, "CICS1", null, null);
}
catch (IOException ioEx) {
    ioEx.printStackTrace();
}
catch (EPIException epiEx) {
    epiEx.printStackTrace();
}
```

Exceptions: As the examples show, you must catch exceptions, irrespective of which method you use to construct a basic terminal.

Adding an terminal: There are two ways to construct an extended terminal:

- Using the default constructor
- Using the extended terminal constructor

Default terminal constructor

To create a terminal using the default constructor, first instantiate a terminal, and then use the appropriate set methods on that object. As with the basic terminal, only the setGateway method is mandatory. The setDeviceType, setNetName, setSession and setServer methods are optional as are the methods that set the extended terminal properties. The following setters define the properties for the extended terminal. Using any of these setters implies that you are creating an extended terminal:

- setSignonCapability (Default = sign-on capable, but see "Specifying terminal Sign-on Capability" on page 21)
- setUserid (Default = null)
- setPassword (Default = null)
- setReadTimeout (Default = 0)
- setEncoding (Default = null)
- setInstallTimeout (Default = 0)

```
try {
    EPIGateway eGate = new EPIGateway("tcp://MyGateway",2006);
    Terminal term = new Terminal();
    term.setGateway(eGate);
    term.setServerName("CICS1");
    term.setSignonCapability(Terminal.EPI_SIGNON_INCAPABLE);
    term.setUserid(userid);
    term.setPassword(password);
    term.connect();
}
catch (IOException ioEx) {
    ioEx.printStackTrace();
}
catch (EPIException epiEx) {
    epiEx.printStackTrace();
}
```

After you have defined your terminal, you can use the connect method to install it on CICS (see "Installing a terminal on CICS" on page 62).

Extended terminal constructor

The extended terminal constructor sets all required properties at construction time:

```
try {
    EPIGateway eGate = new EPIGateway("tcp://MyGateway",2006);
    Terminal term = new Terminal(eGate, "CICS1", null, null,
                                 Terminal.EPI_SIGNON_INCAPABLE, userid,
                                 password,0, null);
```

```
        term.connect();
    }
    catch (IOException ioEx) {
        ioEx.printStackTrace();
    }
    catch (EPIException epiEx) {
        epiEx.printStackTrace();
    }
```

Unlike the basic terminal constructor the extended terminal constructor does not automatically install the terminal on CICS. This must be done explicitly using one of the connect methods described below.

Installing a terminal on CICS: The connect methods that can be used are as follows:

Connect()
: This method installs a terminal on CICS using the session property and install timeout property.

Connect(installTimeout)
: This method installs the terminal on CICS using the session property, but updates the install timeout property to that supplied.

Connect(Session, installTimeout)
: This method installs the terminal on CICS, updating the current session property with the supplied session object, and updating the install timeout property with that supplied. Sessions are discussed in "Synchronization and sessions" on page 64.

Deleting terminals

Use the disconnect method to delete terminals from CICS. Ensure that all terminals are deleted without errors before your application ends. To purge a terminal while a transaction is still running, set the PurgeOnDisconnect property to true.

```
term.setPurgeOnDisconnect(true);
term.disconnect();
```

After you have deleted the terminal from CICS, you can install it again by issuing one of the connect() methods:

```
term.disconnect();

.....

term.connect();
```

The Session parameter does not apply to a disconnect call. Deleting a terminal is a synchronous operation.

Starting a transaction

After you have added a terminal to CICS, you can use on of the send methods to start a new transaction.

```
try {
    term.send("EP01",null);
}
catch (EPIException ex) {
    ex.printStackTrace();
}
```

You can also start a transaction by building a screen and sending it to CICS. Screen manipulation and fields are discussed in "Accessing fields on CICS 3270 screens." The following example shows how to start a transaction using the Screen and Field objects:

```
try {
    Screen scr = term.getScreen();
    Field fld = scr.field(1);
    fld.setText("EP01");
    term.send();
}
catch (EPIException ex) {
    ex.printStackTrace();
}
```

Sending and receiving data

Accessing fields on CICS 3270 screens: When a terminal connection to CICS has been established, the Terminal, Screen and Field objects are used to navigate through the screens presented by the CICS server application, reading and updating screen data as required.

The Screen object is created by the Terminal object and is obtained via the getScreen method on the Terminal object. It provides methods for obtaining general information about the 3270 screen, for example, cursor position, and for accessing individual fields by row and column, screen position, or index. The following example prints out field contents, ends the CICS transaction by returning PF3, and disconnects the terminal:

```
// Get access to the Screen object
Screen screen = terminal.getScreen();

for ( int i=1; i <= screen.fieldCount(); i++ ) {
    Field field = screen.field(i); // get field by index
    if ( field.textLength() > 0 )
        System.out.println( "Field " + i + ": " + field.getText() );
}

// Return PF3 to CICS
screen.setAID( AID.PF3 );
```

```
terminal.send();

// Disconnect the terminal from CICS
terminal.disconnect();
```

The Field class provides access to the text and attributes of an individual 3270 field. You can use these in a variety of ways to locate and manipulate information on a 3270 screen:

```
for ( int i=1; i <= screen.fieldCount(); i++ ) {
    Field field = screen.field(i); // get field by index

    // Find unprotected (i.e. input) fields
    if ( field.inputProt() == Field.unprotect )
        ...
    // Find fields the same as a specific text string
    if ( field.getText().equals( "CICS Sign-on") )
        ...
    // Find red fields
    if ( field.foregroundColor() == Field.red )
        ...
}
```

Synchronization and sessions: The Terminal class supports both synchronous and asynchronous sends to the CICS Server. In the case of an asynchronous send, the Screen object is updated while information is being received from the server.

To select synchronous mode, you can either specify *null* for the session using the setSession method, or specify a null session when invoking send. Alternatively, you can implement the session interface and specify that it is a synchronous session.

To select asynchronous mode, implement the session interface and specify that it is an asynchronous session.

Implementing the session interface: You can set the session on a terminal by either using the setSession method, or by passing the session object as part of a send or connect method. “null” is also accepted as a session, meaning that you have no listening object in place for replies and exceptions, and that all calls are synchronous.

The session interface defines two methods that must be implemented: **getSyncType** and **handleReply**. The following code shows a sample implementation:

```
import com.ibm.ctg.epi.*;
public class myASession implements Session {
    public int getSyncType() {
        return Session.async;
    }
    public void handleReply(TerminalInterface term) {
        System.out.println(
          "Reply received Terminal state = " + term.getState());
    }
    public void handleException(TerminalInterface a, Exception e) {
       System.out.println("Exception received:" + e.getMessage());
    }
}
```

This example defines the session as an asynchronous session, because it returns Session.async on the getSyncType call. To make the session a synchronous session, you return Session.sync.

The example shows the **handleReply** and **handleException** methods:

handleReply

You must implement the **handleReply** method. It is called for each transmission received from CICS. Depending on the design of the CICS server program, a Terminal send call can result in one or more replies. The Terminal state property indicates whether the server has finished sending replies:

Terminal.server

Indicates that the CICS server program is still running and has further data to send. The client application can process the current screen contents immediately, or simply wait for further replies. The application cannot delete the terminal, or send the screen to CICS, or start a new transaction.

Terminal.client

Indicates that the CICS server program is now waiting for a response. The client application should process the screen contents and send a reply. The application cannot delete the terminal or start a new transaction.

Terminal.idle

Indicates that the CICS server program has completed. The client application should process the screen contents and either delete the terminal or start a further transaction.

Terminal.failed

Indicates that the transaction has failed to start or complete for some reason, for example, a conversion transaction has timed-out waiting for a response from the application. Invoke the **endReason** and **endReasonString** methods for more information.

Terminal.discon
Indicates that the terminal has been deleted. Invoke the **endReason** and **endReasonString** methods for more information.

Terminal.error
Indicates that the terminal is in error state and cannot be used. Try to delete the terminal to ensure that all terminal resources are cleaned up.

Most Java Client applications wait until the CICS server program has finished sending data (that is, the Terminal state is client or idle) before processing the screen. However, some long-running server programs might send intermediate results or progress information that can usefully be accessed while the Terminal state is still server.

The implementation of the **handleReply** method can read and process data from the Screen object, update fields as required, set the cursor position and AID key in preparation for the return transmission to CICS, and then use the Terminal send method to drive the server application.

In synchronous mode, **handleReply** executes on the same thread that invoked the send. In asynchronous mode, **handleReply** executes on a separate thread.

Note: The **handleReply** method should never attempt to delete a terminal. The disconnect call might make the application hang if called from **handleReply**.

handleException

The **handleException** method is not specified as part of the session interface and is optional unless you are using asynchronous mode sends, when it must be implemented. The compiler does not force implementation of the method. The Terminal class invokes this method if it is present in the Session object.

It is recommended that you also implement the **handleException** method for synchronous mode sends with Automatic Transaction Initiation (ATI) enabled.

For the **handleReply** method, the Terminal state property shows information about the terminal connection.

Exceptions are passed in the Exception object. See “Exception handling” on page 67, for a list of the exceptions that can occur.

ATIs and Read Timeouts: ATI events and Read Timeout events are asynchronous and can occur at any time during the execution of an application, providing ATIs are enabled and a Read Timeout value was specified when creating an extended terminal. If you plan to use these features, it is recommended that you use an asynchronous session. However, these features can be used on a synchronous session; in this case, if any events occur while blocked, **handleReply** runs on the thread that invoked send or disconnect. If your application is not within a send or disconnect invocation, **handleReply** executes on a separate thread.

Exception handling

EPI exceptions can occur when a user application interacts with a terminal. The exception hierarchy is shown in Figure 5.

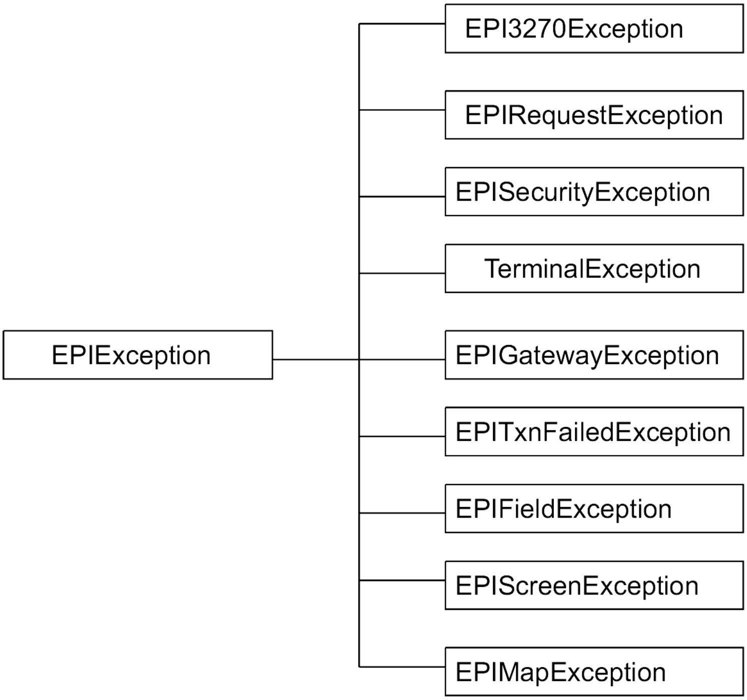


Figure 5. Exception hierarchy

A description of each of these exceptions is given in the Javadoc supplied with the product.

The other type of exception that can occur is IOException.

Use the getErrorCode method to retrieve the exception-specific error code which identifies the exceptions.

If you are using either a null session or a synchronous session, and you have not enabled ATIs and are not using Read Timeouts, all exceptions are thrown on the application thread. When trying to invoke methods such as connect, send, or disconnect, wrap the call in a try/catch/finally block.

When using asynchronous sessions, a problem arises if you have ATIs, or Read Timeouts, or both, enabled. In this case, exceptions can occur while within connect, send, and disconnect method invocations but also outside these calls.

If you use asynchronous sessions, exceptions cannot be thrown on any of the application threads. If you enable ATIs, or Read Timeouts, or both, it is recommended that you use asynchronous sessions.

To know when an exception has occurred when you are not invoking a terminal method, you can implement the **handleException** method on the session. See “Synchronization and sessions” on page 64, for an example of this. You can implement it for both synchronous and asynchronous sessions. If the terminal is unable to throw the exception on the application thread (that is, it is not blocked on a synchronous call or it is an asynchronous session), this method is invoked on a separate thread and the exception is passed to it.

terminal encoding property

You can specify the encoding in which the resulting 3270 data stream is to be constructed. When the terminal is installed, the CICS server (providing it supports EPI Version 2) is informed of the encoding applied to the 3270 data stream. If you specify null, the encoding used by the CICS Transaction Gateway server is used (or the default encoding of the application if the local gateway is being used).

Basic terminals always work in the encoding used by the CICS Transaction Gateway server (or the default encoding of the application if the local gateway is being used).

Refer to your CICS Server document for more information on supported code pages.

Converting BMS maps and using the Map class

A large proportion of existing CICS applications use BMS maps for 3270 screen output. This means that the server application can use data structures corresponding to named fields in the BMS map rather than handling 3270 data streams directly. The EPI BMS conversion utility uses the information in the BMS map source to generate classes specific to individual maps, which allow fields to be accessed by their names.

The utility generates Java classes that applications can use to access the map data as named fields within a map object. A class is defined for each map, allowing field names and lengths to be known at compile time. The generated classes extend the class **Map**, which provides general functions required by all map classes.

Run the BMS map converter utility on the BMS source as follows:

```
java com.ibm.ctg.epi.BMSMapConvert -p package filename.BMS
```

The utility generates .java files containing the source for the map classes. Use the -p parameter to specify the package to put the new files into. This saves you having to edit the files to add the ″package″ statement.

After you have used the EPI BMS utility to generate the map class, use the base EPI classes to reach the required 3270 screens in the usual way. Then use the map classes to access fields by their names in the BMS map. The map classes are validated against the data in the current Screen object.

Using Map classes: The classes generated by the BMS Conversion Utility have the following features:

- The class name is derived from the map name in the BMS source.
- The class extends Map.
- Two constructors are provided. One constructor takes a Screen parameter and throws an EPIException, if the screen has not been produced by the relevant BMS map. The no argument constructor creates a Map that can be validated against a screen later by using the setScreen method.
- The method field provides access to fields in the map, using the BMS source field names (provided as constants within the class).

To use the generated Map class, create a Terminal and start a transaction as usual:

```
try {
   EPIGateway epi = new EPIGateway("jgate", 2006 );
   // Connect to CICS server
   Terminal terminal = new Terminal( epi, "CICS1234", null, null );
   // Start transaction on CICS server
   terminal.send( null, "EPIC", null );
    MAPINQ1Map map = new MAPINQ1Map( terminal.getScreen() );
    Field field;
    // Output text from "PRODNAM" field
    field = map.field(MAPINQ1Map.PRODNAM);
    System.out.println( "Product Name: " + field.getText() );
    // Output text from "APPLID" field
    field = map.field(MAPINQ1Map.APPLID);
```

```
            System.out.println( "Applid : " + field.getText() );
        } catch (Exception exception) {
            exception.printStackTrace();
        }
```

In this example the server program uses a BMS map for its first panel, for which a map class ″MAPINQ1Map″ has been generated. When the map object is created, the constructor validates the screen contents with the fields defined in the map. If validation is successful, fields can then be accessed using their BMS field names instead of by index or position from the Screen object:

BMS Map objects can also be used within the Session **handleReply** method.

For validation to succeed, the entire BMS map must be available on the current screen. A map class cannot therefore be used when some or all of the BMS map has been overlaid by another map or by individual 3270 fields.

EPIRequest

To make EPI type calls to CICS you need to create EPIRequest objects. For more information on these objects, refer to the Javadoc supplied with the CICS Transaction Gateway. Note that CICS Transaction Gateway for z/OS does not support EPIRequest objects.

Using the EPIRequest class

It is recommended that you use the EPI support classes or the J2EE EPI resource adapter if you are writing programs to interface with CICS 3270 transactions, because support for the EPIRequest class might be removed in a future release of the CICS Transaction Gateway. However, read this section if you intend to use the EPIRequest class.

When a Java Client application connects to CICS using EPI, the application appears to CICS as a 3270 terminal. It is, therefore, important to be aware of the 3270 data streams that might flow in both directions. After an event has been returned to a Java application, the size field of the EPIRequest object indicates the size of the data array returned.

It is also important to be aware of the principles and restrictions governing EPI programming, and to be aware that there might be minor differences in the working of the EPI code on different operating systems. For example, if you are running a CICS Transaction Gateway on Windows, you will probably need to send Transaction identifiers in the data array of the EPIRequest object, rather than in the EPIRequest object’s Transid field.

When getting events from CICS it is recommended that you use the EPI_WAIT option, and ensure that the size field of the EPIRequest object is set to the maximum size of the 3270 data stream that CICS might return.

Parameter lengths: When using the EPIRequest class it is important to note that the parameters have maximum lengths. Any parameters passed exceeding these lengths will be truncated.

Generally, EPI programs written using the CICS Transaction Gateway should:

1. Open a connection to the Gateway.
2. Add a terminal.
3. Start a transaction.
4. Get an event until one of the following happens:
   - the event received is an end transaction or a converse
   - a severe error is received
5. If the event received is a converse, send the reply and return to the get event loop.
6. If the event received is an end transaction, delete the terminal and do a last get event to obtain the end terminal event.
7. Close the connection to the Gateway.

Terminal Indexes

For remote connections, terminal indexes can only be used on the connection to which they were assigned. See "EPI security" for more information. For local connections, all local JavaGateways can access terminal indexes on other local JavaGateways, provided they are in the same JVM.

EPI null stripping

EPI null stripping is an internal optimization of the CICS Transaction Gateway that reduces the amount of data flowed from the Gateway daemon to the Gateway classes. If you run Java Client applications in remote mode, use the latest version of the Gateway classes and the Gateway daemon to benefit from EPI null stripping.

EPI and z/OS

The EPI classes are not available for z/OS. If you want to run transactions in the manner of the EPI, you should use the ECI and set up a request for DFHWBTTA. This is described in the *CICS Internet guide*.

EPI security

Terminal ids can only be used on the same JavaGateway that created the terminal. Again, this is a security feature to stop other programs that connect to the same CICS Transaction Gateway from manipulating that terminal.

Making External Security Interface Calls from a Java Client program

Use the ESIRequest base class for password management.

Table 11 shows Java objects corresponding to the ESI terms listed in "Input and output information for ESI functions" on page 25.

The ESIRequest class is not available for z/OS.

Table 11. ESI terms and corresponding Java objects

ESI term	Java object
Current password	ESIRequest.setCurrentPassword()
New password	ESIRequest.setNewPassword()
Server name	ESIRequest.setServer()
User ID	ESIRequest.setUserid()

Verifying a password using ESI

Use the verifyPassword method, passing the current password, user ID and server name to verify a password.

Changing a password using ESI

Use the changePassword method, passing the current password, new password, user ID and server name to change a password.

Compiling and running a Java Client application

Performance issues

There are several performance issues to consider when you run Java client applications. The Java Virtual Machine (JVM) allocates a fixed size of stack space for each running thread in an application. You can usually control the amount of space that Java allocates by setting limits on the following sizes:

- The native stack size, allocated when running native JIT (Just-In-Time) compiled code.
- The Java stack size, allocated when running Java Bytecode.
- The initial Java heap size.
- The maximum Java heap size.

How you set these limits depends on your JVM. Refer to your Java documentation for more information.

Setting up the CLASSPATH

Before you write any Java client programs, update the CLASSPATH environment variable to include the jar files supplied with CICS Transaction Gateway. For example, on Windows:

```
CLASSPATH = <install_path>\classes\ctgclient.jar;
            <install_path>\classes\ctgserver.jar
```

The ctgserver.jar file is required in CLASSPATH only for JavaGateways using the local URL.

For more information on setting CLASSPATH, see the *CICS Transaction Gateway: Administration* book for your operating system.

Using a browser and CICS Transaction Gateway on the same workstation

If you intend to use a browser and CICS Transaction Gateway on the same workstation, remove ctgclient.jar and ctgserver.jar from the CLASSPATH setting. If you do not remove them, you are likely to receive the following error when running applications:

```
ERROR: java.io.IOException:
CTG6664E: Unable to load relevant class to support the tcp protocol
```

The reason for the error is that Java searches the CLASSPATH environment variable before downloading classes across the network. If the required class is local, Java attempts to use it. However, using class files from the local file system breaks Java application security rules; therefore an exception is raised.

Problem determination for Java Client programs

Tracing in Java client programs

You can control tracing in Java client programs using:

- calls to the com.ibm.ctg.client.T class

  For example, from within a user application:

  ```
  if (getParameter("trace") != null)
   {
     T.setOn(true);
   }
  ```

  where `trace` is a startup parameter that can be set on the user program.
- Gateway.T system properties

  For example:

  ```
  java -Dgateway.T=on com.usr.smp.test.testprog1
  ```

  which specifies full debug for `testprog1`.

  For more information on the use of system properties, refer to your Java documentation.

It is recommended that applications implement an option to turn trace on.

The following is an explanation of the various trace levels available. The names of calls and properties are case sensitive.

Trace level
Standard

com.ibm.ctg.client.T call
T.setOn (true/false)

System property
gateway.T.trace=on

Definition
The standard option for application tracing.

By default, it displays only the first 128 bytes of any data blocks (for example the *commarea*, or network flows).

This trace level is equivalent to the Gateway trace set by the `ctgstart –trace` option.

Trace level
Full Debug

com.ibm.ctg.client.T call
T.setDebugOn (true/false)

System property
gateway.T=on

Definition
The debugging option for application tracing.

By default, it traces out the whole of any data blocks. The trace contains more information about the CICS Transaction Gateway than the standard trace level.

This trace level is equivalent to the Gateway debug trace set by the `ctgstart –x` option.

Trace level
Exception Stacks

com.ibm.ctg.client.T call
T.setStackOn (true/false)

System property
gateway.T.stack=on

Definition
The exception stack option for application tracing.

It traces most Java exceptions, including exceptions which are expected during normal operation of the CICS Transaction Gateway. No other tracing is written.

This trace level is equivalent to the Gateway stack trace set by the `ctgstart –stack` option.

You can further configure the tracing by using the following options:

com.ibm.ctg.client.T call
T.setTFile(true,*filename*)

System property
gateway.T.setTFile=*filename*

Option usage
The value *filename* specifies a file location for writing of trace output. This is as an alternative to the default output on stderr. Long file names must be surrounded by quotation marks, for example: ″trace output file.log″

com.ibm.ctg.client.T call
T.setTruncationSize(*number*)

System property
gateway.T.setTruncationSize=*number*

Option usage
The value *number* specifies the maximum size of any data blocks that will be written in the trace. Any positive integer is valid. If you specify a value of *0*, then no data blocks will be written in the trace. If a negative value is assigned to this option the exception java.lang.IllegalArgumentException will be raised.

com.ibm.ctg.client.T call
T.setDumpOffset(*number*)

System property
gateway.T.setDumpOffset=*number*

Option usage
The value *number* specifies the offset from which displays of any data blocks will start. If the offset is greater than the total length of data to be displayed, an offset of *0* will be used. If a negative value is assigned to this option the exception java.lang.IllegalArgumentException will be raised.

com.ibm.ctg.client.T call
T.setTimingOn (true/false)

System property
gateway.T.timing=on

Option usage
Specifies whether or not to display time-stamps in the trace.

Use the options in addition to one of the directives to switch tracing on.

For example, the following switches standard tracing on, and sets the maximum size of any data blocks to be dumped to 20 000 bytes:

```
java -Dgateway.T.trace=on -Dgateway.T.setTruncationSize=20000
```

Security for Java Client programs

CICS Transaction Gateway security classes

The CICS Transaction Gateway provides the following classes (security exits) for implementing security:

com.ibm.ctg.security.JSSEServerSecurity
: Use this interface to allow the exposure of of X.509 Client Certificates when using the JSSE protocol.

  Refer to your JSSE, or Java, documentation for information on using X.509 certificates.

com.ibm.ctg.security.ServerSecurity
: Use this interface for server-side security classes that do not require the exposure of SSL Client Certificates.

com.ibm.ctg.security.ClientSecurity
: Use this interface for all client-side security classes.

com.ibm.ctg.util.RACFUserid
: This class tries to map an X.509 Client Certificate to a RACF userid. The certificate must already be associated with a valid RACF userid.

The **JSSEServerSecurity** and **ServerSecurity** interfaces and partner **ClientSecurity** interface define a simple yet flexible model for providing security when using CICS Transaction Gateway. Implementations of the interfaces can be as simple, or as robust, as necessary; from simple XOR (exclusive-OR) scrambling to use of the Java Cryptography Architecture.

The JSSEServerSecurity interface has been designed to work in conjunction with the Secure Sockets Layer (SSL) protocol. The interface allows server-side security objects access to a Client Certificate passed during the initial SSL handshake. The exposure of the Client Certificate depends on the the CICS Transaction Gateway being configured to support Client Authentication.

An individual JavaGateway instance has an instance of a ClientSecurity class associated with it, until the JavaGateway is closed. Similarly, an instance of the partner JSSEServerSecurity or ServerSecurity class is associated with the connected Java client, until the connection is closed.

The basic model consists of:

- An initial handshake to exchange pertinent information. For example, this handshake could involve the exchange of public keys. However, at the interface level, the flow consists of a simple byte-array, therefore an implementation has complete control over the contents of its handshake flows.
- The relevant ClientSecurity instance being called to encode outbound requests, and decode inbound replies.
- The partner JSSEServerSecurity or ServerSecurity instance, being called to decode inbound requests and to encode outbound replies.

  The inbound request, and Client Certificate, is exposed via the **afterDecode()** method. For JSSE, the afterDecode() method exposes the GatewayRequest object, along with the **javax.security.cert.X509Certificate[]** certificate chain object.

ClientSecurity, JSSEServerSecurity, or ServerSecurity class instances should maintain as data members sufficient information from the initial handshake to correctly encode and decode the flows. At the server, each connected client has its own instance of the ServerSecurity implementation class.

Using a Java 2 Security Manager

Java 2 provides a Security Manager system that controls access to Java resources. It restricts access to Java resources by using a security policy. Examples of protected resources are: reading a file, and opening a network socket. When a program tries to access a protected resource, the Java Security Manager verifies that both the code trying to access the resource, and, possibly, the caller of that code, have appropriate permissions. Without these permissions, the program cannot run.

If you are using any of the CICS Transaction Gateway Java APIs under a Java 2 security environment (such as a J2EE server), your application needs Java permissions to execute correctly. The only exception to this is if you are using the J2EE APIs in a managed environment.

Figure 6 on page 78 shows the minimum permissions that your application needs to use Gateway Java APIs. It might need additional permissions to execute correctly.

```
java.net.SocketPermission "*", "resolve";
java.util.PropertyPermission "*", "read";
java.io.FilePermission "${user.home}${file.separator}ibm${file.separator}ctg${file.separator}-",
 "read,write,delete";
java.lang.RuntimePermission "loadLibrary.*", "";
java.lang.RuntimePermission "shutdownHooks", "";
java.lang.RuntimePermission "modifyThread", "";
java.lang.RuntimePermission "modifyThreadGroup", "";
java.lang.RuntimePermission "readFileDescriptor", "";
java.lang.RuntimePermission "writeFileDescriptor", "";
java.security.SecurityPermission "putProviderProperty.IBMJSSE", "";
java.security.SecurityPermission "insertProvider.IBMJSSE", "";
java.security.SecurityPermission "putProviderProperty.IBMJCE", "";
java.security.SecurityPermission "insertProvider.IBMJCE", "";
javax.security.auth.PrivateCredentialPermission "* * \"*\"","read";
java.lang.RuntimePermission "accessClassInPackage.sun.io", "";
```

Figure 6. Required Java 2 Security Manager permissions

Permissions to access the file system

Depending on the functions performed by your program, the CICS Transaction Gateway Java APIs might require access to the file system, for example to write trace files. The following permission gives permission for the CICS Transaction Gateway classes to access the file system on UNIX and Linux systems:

```
permission java.io.FilePermission "${user.home}${file.separator}ibm
${file.separator}ctg${file.separator}-","read,write,delete";
```

The format of the permission might vary depending on the installation, and you can specify alternative locations, or none at all. CICS Transaction Gateway classes require access to the file system in the following cases:

- For writing trace information to a file
- For accessing key rings, if you are using JSSE for your SSL protocol implementation

  See *Network security*, in the *CICS Transaction Gateway: Administration* book for your operating system, for information on how JSSE is selected as the implementation.

For example, on Windows, you can specify the permission:

```
permission java.io.FilePermission "c:\trace\-",
                                  "read,write,delete";
```

to allow access to the directory c:\trace and all subdirectories.

Or, for example, on UNIX and Linux systems, you can specify the permission:

```
permission java.io.FilePermission "/tmp/ibm/",
                                  "read,write,delete";
```

to allow access to the directory /tmp/ibm and all subdirectories.

Chapter 7. Programming using the J2EE Connector Architecture

This information describes how to program using the ECI and EPI resource adapters provided by the CICS Transaction Gateway.

Overview of the programming interface of the J2EE Connector Architecture

The purpose of the J2EE Connector Architecture (JCA) is to connect Enterprise Information Systems (EISs), such as CICS, into the J2EE platform. The JCA offers a number of qualities of service which can be provided by a J2EE application server. These qualities of service include security credential management, connection pooling and transaction management.

These qualities of service are provided by means of system level contracts between a resource adapter provided by the CICS Transaction Gateway, and the J2EE application server. There is no need for any extra program code to be provided by the user. Thus the programmer is free to concentrate on writing the business code and need not be concerned with providing quality of service.

The JCA defines a programming interface called the Common Client Interface (CCI). This interface can be used, with minor changes, to communicate with any EIS. The CICS Transaction Gateway provides resource adapters which implement the CCI for interactions with CICS.

The Common Client Interface (CCI)

The CCI is a high level interface defined by the JCA and is available to J2EE developers using the External Call Interface (ECI) and the External Presentation Interface (EPI) to communicate with programs running on a CICS server. There is no resource adapter CCI for the External Security Interface (ESI).

The CCI has two distinct class types:

Generic CCI classes

Generic CCI classes are used to request a connection to an EIS such as CICS, and execute commands on that EIS, passing input and retrieving output. These classes are generic in that they do not pass information that is specific to a particular EIS. Connection and ConnectionFactory are examples of generic CCI classes.

CICS-specific CCI classes

CICS-specific classes are used to pass specific information between the

Java Client application and CICS. ECIInteractionSpec and ECIConnectionSpec are examples of CICS-specific classes.

The programming interface model

Applications using the CCI have a common structure, independent of the EIS that is being used. The JCA defines Connections and ConnectionFactories which represent the connection to the EIS. These objects allow a J2EE application server to manage security, transaction context, and connection pools for the resource adapter.

An application must start by obtaining a ConnectionFactory from which a Connection can be obtained. The properties of this Connection can be overridden by a ConnectionSpec object. The ConnectionSpec class is CICS-specific, so may be either an ECIConnectionSpec or an EPIConnectionSpec.

After an connection has been obtained, an Interaction can be created from the Connection in order to make a particular request. As with the Connection, Interactions can have custom properties set by the CICS-specific InteractionSpec class (ECIInteractionSpec or EPIInteractionSpec). To perform the Interaction, call the execute() method and use CICS-specific Record objects to hold the data. For example:

```
Obtain a ConnectionFactory
Connection c = cf.getConnection(ConnectionSpec)
Interaction i = c.createInteraction()
InteractionSpec is = newInteractionSpec();
i.execute(spec, input, output)
```

The ConnectionFactory can be obtained in two ways:

- If you are using a J2EE application server, the ConnectionFactory is normally created from the resource adapter by means of an administration interface. This ConnectionFactory has custom properties set for it, for example the Gateway to be used would be set as a ConnectionURL. When the ConnectionFactory has been created, it can be made available for use by any enterprise applications through JNDI. This type of environment is called a managed environment. A managed environment allows a J2EE application server to manage the qualities of service of the connections. Refer to the Administration Guide for a description of deployment into a managed environment.
- If you are not using a J2EE application server, you must create a ManagedConnectionFactory and set its custom properties. You can then create a ConnectionFactory from the ManagedConnectionFactory. This type of environment is called a non-managed environment. A non-managed environment does not allow a J2EE application server to manage connections.

Record objects

Record objects are used to represent data passing to and from the EIS. In the case of the ECI, this is a representation of a COMMAREA or channels and containers. In the case of the EPI, it is a terminal screen. A sample Record is provided for the ECI and a Screenable interface is provided for the EPI to access the screen data. It is recommended that application development tools are used to generate these Records.

The ECI resource adapters

The ECI resource adapters provide a high level CCI interface to the ECI that can be used to link to CICS server programs and pass data in COMMAREAs or channels and containers without having to issue ECI requests. The resource adapters can be deployed into a J2EE application server to allow J2EE enterprise applications to access CICS. When the JCA is used, connection pooling, security, and transaction context are managed by the J2EE application server instead of the application.

Two resource adapters are supplied:

CICS Transaction Gateway on z/OS

Adapter cicseciXA.rar supports both XA and local transactions.

CICS Transaction Gateway on z/OS and multiplatforms

Adapter cicseci.rar supports the LocalTransaction interface, and global transactions in local mode under WebSphere.

The cicseciXA.rar resource adapter should be used for two-phase commit functionality with IPIC. For one-phase commit functionality the cicseciXA.rar resource adapter can be used, however performance might be improved by using cicseci.rar resource adapter.

See “Transaction management” on page 89 for details of the transaction management models that each resource adapter supports.

The EPI resource adapter

The EPI resource adapter provides a high level CCI interface to the EPI which can be used to install terminals and run 3270-based transactions on a CICS server. There is no support for Automatic Transaction Initiation (ATI). The resource adapter can be deployed into a J2EE application server to allow J2EE enterprise applications to access CICS. When the JCA is used, connection pooling, security, and transaction context are managed by the J2EE application server instead of the application.

Managed and non-managed environments

A Java Client application using J2EE can run in one of two different environments:

- Managed environment

A managed environment is one in which a J2EE application server such as theWebSphere Application Server performs management of connections, transactions, and security, thus relieving the application developer of the necessity to produce code for this.

- Non-managed environment

  A non-managed environment is one in which the application uses the resource adapters directly without the intervention of a J2EE application server. In this case the application must contain code to handle management of connections, transactions and security.

The Common Client Interface

The Common Client Interface (CCI) of the J2EE Connector Architecture provides a standard interface that allows developers to communicate with any number of Enterprise Information Systems (EISs) through their specific resource adapters, using a generic programming style. The CCI is closely modeled on the client interface used by Java Database Connectivity (JDBC), and is similar in its idea of Connections and Interactions.

Generic CCI Classes

The generic CCI classes define the environment in which a J2EE component can send and receive data from an EIS. When you are developing a J2EE component you must implement the following steps:

1. Use the ConnectionFactory object to create a Connection object.
2. Use the Connection object to create an Interaction object.
3. Use the Interaction object to execute commands on the EIS.
4. Close the Interaction and Connection.

The following example shows the use of the J2EE CCI interfaces to execute a command on an EIS.

```
ConnectionFactory cf = <Lookup from JNDI namespace>
Connection conn = cf.getConnection();
Interaction interaction = conn.createInteraction();
interaction.execute(<Input output data>);
interaction.close();
conn.close();
```

CICS-specific classes

TheCICS Transaction Gateway resource adapters provide additional classes specific to CICS. The following object types are used to define the ECI- and EPI-specific properties:

- InteractionSpec objects
- ConnectionSpec objects

Spec objects define the action that a resource adapter carries out, for example by specifying the name of a program which is to be executed on CICS.

Record objects store the input/output data that is used during an interaction with an EIS, for example a byte array representing an ECI COMMAREA.

The following example shows a complete interaction with an EIS. In this example input and output Record objects and Spec objects are used to define the specific attributes of both the interaction and the connection. The example uses setters to define any component-specific properties on the Spec objects before they are used.

```
ConnectionFactory cf = <Lookup from JNDI namespace>
ECIConnectionSpec cs = new ECIConnectionSpec();
cs.setXXX();    //Set any connection specific properties

Connection conn = cf.getConnection( cs );
Interaction interaction = conn.createInteraction();
ECIInteractionSpec is = new ECIInteractionSpec();
is.setXXX();    //Set any interaction specific properties

RecordImpl in = new RecordImpl();
RecordImpl out = new RecordImpl();

interaction.execute( is, in, out );
interaction.close();
conn.close();
```

The following sections cover the ECI and EPI implementations of the CCI classes in detail.

Using the ECI resource adapters

The details in this topic apply to both resource adapters (cicseci.rar and cicseciXA.rar).

The ECI resource adapters allow a J2EE developer to access CICS programs, using COMMAREAs and channels to pass information to and from the server. Table 12 shows the JCA objects corresponding to the ECI terms listed in "Input and output information for external calls to CICS" on page 9. The CCI interfaces for CICS are in the `com.ibm.connector2.cics` package.

Table 12. ECI terms and corresponding JCA objects

ECI term	JCA object: property
Abend code	CICSTxnAbendException
COMMAREA	Record
Channel	ECIChannelRecord or MappedRecord

Table 12. ECI terms and corresponding JCA objects (continued)

ECI term	JCA object: property
Container with a data type of BIT	byte[]
Container with a data type of CHAR	String
ECI timeout	ECIInteractionSpec:ExecuteTimeout
LUW identifier	J2EE transaction
Password	ECIConnectionSpec:Password
Program name	ECIInteractionSpec:FunctionName
Server name	ECIConnectionFactory:ServerName
SocketConnectTimeout	ECIConnection:SocketConnectTimeout
TPNName	ECIInteractionSpec:TPNName
TranName	ECIInteractionSpec:TranName
User ID	ECIConnectionSpec:UserName

Introduction to channels and containers

Channels and containers provide a method of transferring data between CICS programs, in amounts that far exceed the 32KB limit that applies to communication areas (COMMAREAs).

Each container is a ″named COMMAREA″ that is not limited to 32KB. Containers are grouped together in sets called channels.

The channel/container model has several advantages over the communication areas (COMMAREAs) used by CICS programs to exchange data:

- Unlike COMMAREAs, channels are not limited in size. There is no limit to the number of containers that can be added to a channel, and the size of individual containers is limited only by the amount of storage that you have available. Consider the amount of storage available to other applications when you create large containers.
- Because a channel can comprise multiple containers, it can be used to pass data in a more structured way, allowing you to partition your data into logical entities. In contrast, a COMMAREA is a monolithic block of data.
- Unlike COMMAREAs, channels do not require the programs that use them to know the exact size of the data returned.
- Channels can be used by CICS application programs written in any of the CICS-supported languages. For example, a Java client program on one CICS region can use a channel to exchange data with a COBOL server program on a back-end AOR.
- CICS automatically destroys containers (and their storage) when they go out of scope.

There are also implications when using channels and containers in preference to COMMAREAs:

- A channel can use more storage than a COMMAREA designed to pass the same data. This is because:
  1. Container data can be held in more than one place.
  2. COMMAREAs are accessed by pointer, whereas the data in containers is copied between programs.

For more information on using channels and containers within the JCA framework, see "Using the ECI resource adapters with channels and containers."

For more information on using channels and containers with ECI calls, see "Creating channels and containers for ECI calls" on page 51.

For more information on channels and containers, see the CICS Transaction Server for z/OS Channels learning path.

Using the ECI resource adapters with channels and containers

To use channels and containers in the J2EE Connector Architecture (JCA), use a MappedRecord structure (ECIChannelRecord) to hold your data. When the MappedRecord is passed to the execute() method of ECIInteraction, the method uses the MappedRecord itself to create a channel and converts the entries inside the MappedRecord into containers before passing them to CICS.

The MappedRecord allows multiple data records to pass over the same interface to and from the execute() method of ECIInteraction. A container is created for each entry within the channel. There are two data types of container and you can have a combination of container types in one channel. The containers are the following types:

- A container with a data type of BIT. This type of container is created when the entry is a byte[], or implements the `javax.resource.cci.Streamable` interface. No code page conversion takes place.
- A container with a data type of CHAR. This type of container is created when you use a String to create the entry.

You can create your own data records, which must conform to existing JCA rules (they must implement the `javax.resource.cci.Streamable` and `javax.resource.cci.Record` interfaces). Any data records you create are treated as containers with a data type of BIT.

You can also use an existing Record type, for example, JavaStringRecord, to create a container with a data type of BIT.

The MappedRecord.getRecordName method gets the name of the channel. When creating your Record, you must make sure that the name is not an empty string. The record.getRecordName method retrieves the name of the containers.

The JCA resource adapter handles MappedRecords and Records differently, when it receives the data in the execute() method of ECIInteraction.

- When a MappedRecord is received, the resource adapter uses a channel to send the data.
- When a Record (that is not a MappedRecord) is received, the resource adapter uses a COMMAREA to send the data.

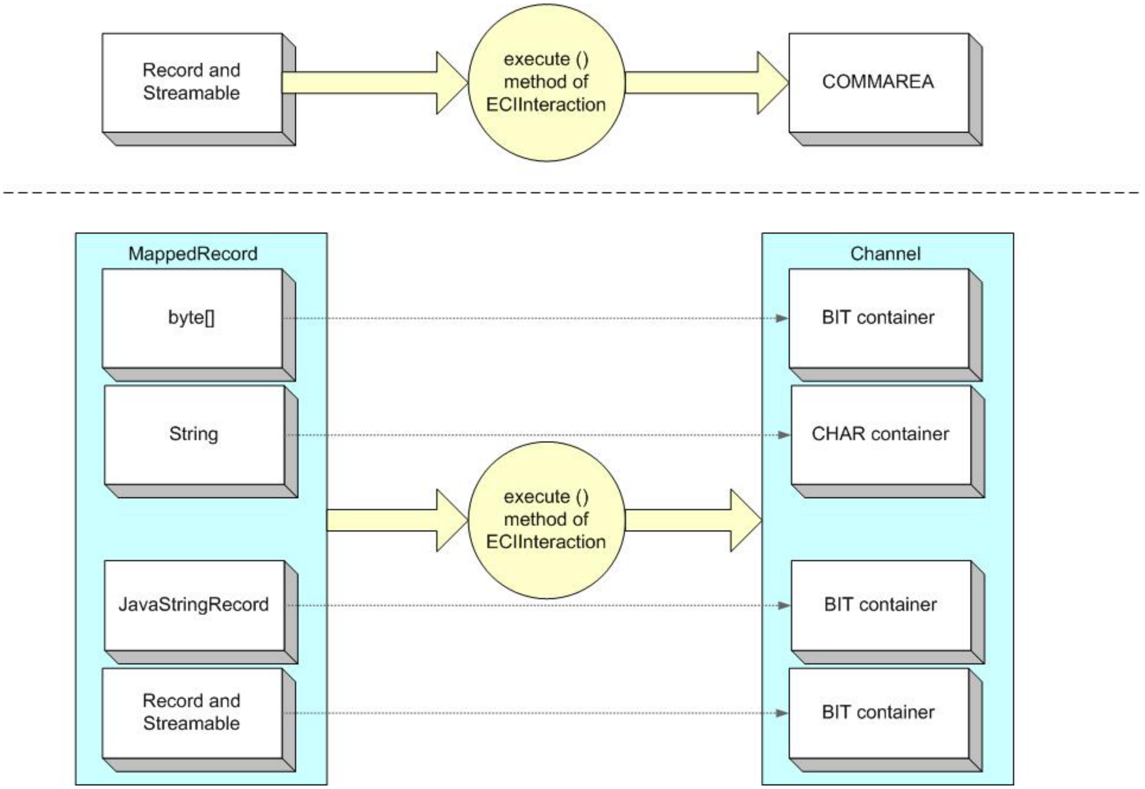


Figure 7. Data conversion by the execute() method of ECIInteraction, depending on whether it receives a Record or MappedRecord

Connecting to a CICS server using the ECI resource adapter

Use the ConnectionFactory and Connection interfaces to establish a connection with a CICS server. The ECI resource adapter provides implementations of the connection interfaces but you should not work directly with the ECI implementations. Use the ECIConnectionSpec class directly to define the properties of the connection.

The ECIConnectionSpec class allows the J2EE component to override the userid and password set at deployment time. So to obtain a connection you code something like this:

```
ConnectionFactory cf = <Lookup from JNDI namespace>
ECIConnectionSpec cs = new ECIConnectionSpec();
cs.setUserName("myuser");
cs.setPassword("mypass");
Connection conn = cf.getConnection(cs);
```

Linking to a program on a CICS server

Use the Interaction interface to link to a server program. The ECI resource adapter provides an implementation of the Interaction interface but you should not use this directly. You should use the ECIInteractionSpec class directly, to define the properties of the interaction:

- Set the FunctionName property to the name of the CICS server program.
- Set the InteractionVerb to SYNC_SEND for an asynchronous call or SYNC_SEND_RECEIVE for a synchronous call. Use SYNC_RECEIVE to retrieve a reply from a asynchronous call.

  Note:

  1. When a SYNC_SEND call has been issued with the execute() method of a particular ECIInteraction object, that instance of ECIInteraction cannot execute another SYNC_SEND, or SYNC_SEND_RECEIVE, until a SYNC_RECEIVE has been executed.
  2. Simultaneous asynchronous calls to the same connection are permitted, provided this does not result in two asynchronous calls being outstanding within the same transaction scope. If this happens an exception is thrown.
  3. If you are using the adapter in local mode with IBM WebSphere Application Server for z/OS, and you require transactional support, specify the SYNC_SEND_RECEIVE interaction type. If you use SYNC_SEND and SYNC_RECEIVE to issue asynchronous requests, the ECI requests will be issued with sync on return, and will be outside the scope of the current global transaction. In remote mode, asynchronous calls work normally.
- If you are using channels and containers, the program receiving the data does not need to know the exact size of the data returned. If you are using COMMAREAs, set the CommareaLength property to the length of the COMMAREA being passed to CICS. If this is not supplied a default is used:

SYNC_SEND, SYNC_SEND_RECEIVE
Length of input record data

SYNC_RECEIVE
The value of ReplyLength

- Set the ReplyLength property to the length of the data stream to be returned from the Gateway daemon to the JCA application. This can reduce the data transmitted over the network if the data returned by CICS is less than the full COMMAREA size, and you know the size of the data in advance.

  The JCA application still receives a full COMMAREA of the size specified in CommareaLength, only the amount of data sent over the network is reduced. This method is equivalent to the setCommareaInboundLength() method available for the ECIRequest class.

  If you do not set ReplyLength, the CICS Transaction Gateway automatically strips trailing zeros from the COMMAREA sent from the Gateway daemon to the JCA application, without needing to know the size of the data in advance.

  For more information on COMMAREA stripping, see "Performance considerations when transmitting data in a COMMAREA" on page 53.

As with ECIConnectionSpec, you can set properties on the ECIInteractionSpec class at either construction time or by using setters. Unlike ECIConnectionSpec, the ECIInteractionSpec class has been designed as a Java bean. So in a managed environment, your server may provide tools to allow you to define these properties using a GUI without writing any code.

- To specify a value for ECI timeout, set the ExecuteTimeout property of the ECIInteractionSpec class to the ECI Timeout value. Allowable values are:

  0 No timeout. This is the default value.

  A positive integer
  Time in milliseconds.

If you are using a CICS Transaction Gateway on z/OS, you cannot specify a value for ECI timeout. As an alternative, you can set the TIMEOUT parameter in the EXCI options table DFHXCOPT. For more information see the *CICS External Interfaces Guide*.

See "Timeout of the ECI request" on page 14 for more information on ECI timeouts.

ECI resource adapter CICS-specific records using the streamable interface

For input and output, the ECI resource adapter supports only records that implement the `javax.resource.cci.Streamable` interface. MappedRecords that are used to make up channels and containers also conform to this interface. The `javax.resource.cci.Streamable` interface allows the ECI resource adapter to read streams of bytes that make up the CICS COMMAREAs or channels and containers directly from, and write them to, the Record objects supplied to the execute() method of ECIInteraction. The following example shows how to build a record for use as input by the ECI resource adapter, using the method supplied in the `javax.resource.cci.Streamable` interface.

```
Byte commarea[] = new byte[10];
ByteArrayInputStream stream = new ByteArrayInputStream(commarea);
Record in = new RecordImpl();
in.read(stream);
int.execute(..., in, ...);
```

To retrieve a byte array from the output record, the reverse of the process shown in the above example can be achieved by using the output records write() method using a ByteArrayOutputStream object as the parameter. The streams toByteArray() method will then provide the CICS COMMAREA or channel and container output in the form of a byte array. In the above example a class called RecordImpl is used as the concrete implementation class of the `javax.resource.cci.Record` interface. To provide more functionality for your specific J2EE components, you can write implementations of the Record interface that allow you to set the contents of the record using the constructor. This avoids the use of the ByteArrayInputStream used in the above example. A managed environment may provide tools that allow you to build implementations of the Record interface that are customized for your J2EE components needs without writing any code.

Transaction management

Two resource adapters are supplied.

cicseci.rar

This resource adapter provides LocalTransaction support when deployed on any supported J2EE application server. Local transactions are not supported when using WebSphere Application Server for z/OS with CICS TG on z/OS in local mode, as the resource adapter provides global transaction support in conjunction with MVS™ RRS.

cicseciXA.rar

This provides both XATransaction and LocalTransaction support when deployed on any supported J2EE application server connecting to a remote CICS TG for z/OS. It also provides global transaction support when using WebSphere Application Server for z/OS with a CICS TG on z/OS in local mode.

The cicseciXA.rar resource adapter should be used for two-phase commit functionality with IPIC. For one-phase commit functionality the cicseciXA.rar resource adapter can be used, however performance might be improved by using cicseci.rar resource adapter.

In order to provide for different transactional qualities of service for J2EE applications, it is possible to deploy both CICS resource adapters into the same J2EE application server. When multiple resource adapters are used in the same J2EE application server, they must all be at the same version.

See *CICS Transaction Gateway: z/OS Administration* for information about installing the resource adapters. See the Migration section for support for resource adapters.

When carrying out multiple interactions with CICS using the ECI resource adapter you might wish to group all actions together to ensure that they either all succeed or all fail. The preferred way is to let the J2EE application server manage this; such transactions are known as *container-managed transactions*. However, to do this yourself use the LocalTransaction or UserTransaction interface. Such transactions are known as *bean-managed transactions*. Bean-managed transactions that use the LocalTransaction interface can group work performed only through the resource adapter; the UserTransaction interface allows all transactional resources within the application to be grouped.

cicseciXA.rar with bean-managed transactions
Supports the UserTransaction and LocalTransaction interfaces.

cicseci.rar with bean-managed transactions
Supports the LocalTransaction interface.

XA overview

A global transaction is a recoverable unit or work performed by one or more resource managers in a distributed transaction processing environment, coordinated by an external transaction manager.

The resources which are updated by the transaction can take many forms such as a database table, a messaging queue or the resources updated by the execution of a CICS transaction. Each of these resources is managed by a resource manager. Where the recoverable resources updated by the global transaction are all managed by the same resource manager, a one-phase-commit protocol is adequate to ensure that all resources are updated in an atomic manner.

However, where the resources updated by a global transaction are managed by multiple resource managers, a two-phase-commit protocol is required. This protocol ensures the atomic nature of the transaction is maintained by ensuring that all resource managers update their resources in a consistent manner. The cicseciXA.rar supports the two-phase-commit XA protocol and enables J2EE applications to include CICS resources in such global transactions.

In both the one-phase-commit and XA scenarios a transaction manager is responsible for controlling the execution of the transaction and coordinating the resource managers to ensure that the transaction executes in an atomic manner.

An example of where this behavior would be required is an online flight booking, which uses one resource manager to debit a customer's bank account and another to reserve the customer a flight. The customer's account must be updated if - and only if - the flight is booked; and vice-versa.

For information on using XA transactions with J2EE applications, see *Redpaper: Transactions in J2EE, REDP-2659-00*.

WebSphere optimizations: The following optimizations are supported:

- Last participant support
- Only-agent optimization

See the documentation supplied with WebSphere Application Server for more details.

MVS image restrictions: When extended mode base Java requests, or J2EE requests are being issued, the CICS Transaction Gateway must be on the same MVS image as the CICS region it is sending requests to. The same restriction also applies to applications that use the local protocol. These applications must be running on the same MVS image as the CICS region they are sending requests to.

Certain restrictions affect where the Gateway daemon and local applications can run:

- When J2EE transactions or extended LUW requests are being issued using EXCI connections, the CICS® Transaction Gateway instance must execute on the same MVS™ image as the CICS region that it is sending requests to.
- Each Gateway daemon that is configured as part of a Gateway group must be run on the same MVS image.
- In a Gateway group, the ctgmaster process and all Gateway daemon instances must use the same HFS installation.
- A maximum of 255 Gateway groups is allowed in an MVS image.

Restrictions on WebSphere Application Server for z/OS

On WebSphere Application Server for z/OS it is not possible to use the local transaction interface if you have configured the ECI resource adapter to run in local mode. In this environment if you plan to connect to CICS using the local protocol, do not attempt to get a LocalTransaction object from the connection (in other words do not invoke the method getLocalTransaction() on your connection object). In managed mode, attempts to invoke getLocalTransaction() will result in a NotSupportedException being thrown. In non managed mode, the results are unpredictable.

Samples

J2EE ECI sample programs are provided in the <install_path>\samples subdirectory and as a deployable EAR file in the <install_path>\deployable subdirectory. Refer to “Resource adapter samples” on page 102, for more information.

Using the EPI resource adapter

The CICS EPI resource adapter allows a J2EE component to communicate with CICS transactions that use 3270 data streams for input and output. The resource adapter makes each EPIConnection object appear to CICS as a 3270 terminal, thus providing access to the CICS 3270 interface. Table 13 shows the J2EE objects corresponding to the EPI terms listed in“Terminal characteristics” on page 19. The CCI interfaces for CICS are in the `com.ibm.connector2.cics` package.

Note: ATIs are not supported.

Table 13. Terminal attributes and corresponding J2EE objects

EPI term	JCA object:property
Code page	EPIConnectionFactory:Encoding
Columns	EPIInteractionSpec:ScreenWidth
Model	EPIInteractionSpec:DeviceType
Install timeout	EPIConnectionFactory:InstallTimeout
Map name	EPIInteractionSpec:MapName
Map set name	EPIInteractionSpec:MapSetName
Netname	EPIConnectionFactory:NetName
Password	EPIConnectionFactory:Password
Read timeout	EPIConnectionFactory:ReadTimeout
Rows	EPIInteractionSpec:ScreenDepth
Server name	EPIConnectionFactory:ServerName
Sign-on capability	EPIConnectionFactory:SignonType
SocketConnectTimeout	EPIConnection:SocketConnectTimeout
Terminal ID	EPIInteractionSpec:TermID

Table 13. Terminal attributes and corresponding J2EE objects (continued)

EPI term	JCA object:property
User ID	EPIConnectionFactory:Userid

Connecting to a CICS server using the EPI resource adapter CCI

Use the ConnectionFactory and Connection interfaces to establish a connection with a CICS server. The EPI resource adapter provides implementations of the connection interfaces but you should not work directly with the EPI implementations. Use the EPIConnectionSpec class directly to define the properties of the connection.

Setting terminal attributes

With J2EE you do not have to add and delete terminals explicitly. You can use the EPIConnectionSpec class to set the following properties:

- User ID
- Password
- Netname
- Model

Starting a transaction

Use the Interaction interface to start a transaction on a CICS server. The EPI resource adapter provides an implementation of the Interaction interface but you should not use this directly. Each Interaction.execute() call must have an EPIInteractionSpec instance associated with it. Use the EPIInteractionSpec class directly, to define the properties of the interaction:

- Set the FunctionName property to the name of the CICS transaction.
- Set the InteractionVerb to one of the following:
  - SYNC_SEND - A synchronous call. It does not unblock until the EPI transaction has sent all the information that would appear on a screen.
  - SYNC_RECEIVE - A synchronous receive. Used to retrieve the current contents of the screen.
  - SYNC_SEND_RECEIVE - A synchronous call.

The EPIInteractionSpec class also allows you to set the following properties:

- The AID key to be sent to CICS. The default value is *enter*.
- The position of the cursor.
- The output attribute type. This allows you to control what will be held in the attribute byte for the field on a returned screen. It applies only to the streamable interface (see "Sending and receiving data" on page 94).

The EPIInteractionSpec class returns the following properties which can be used by the J2EE component:

- Cursor position
- Screen size
- Terminal ID
- Map name
- Mapset name

Closing an EPIInteraction does not affect the state of the connection; the terminal remains connected.

Sending and receiving data

Use records to pass information to the EPI resource adapter and to retrieve information from the resource adapter. Although the EPI resource adapter supports the *Streamable* interface as defined in the Connector Architecture, if you wish to use the Streamable interface you must write your own records, parsing the input stream and generating the output stream correctly. For information about the Stream format see “Stream Format” on page 96.

The EPI resource adapter provides a more efficient way to access information in the form of a record that is ready to use. This is the recommended way to access and send information to a resource adapter.

The Screen model

The EPI resource adapter provides a record that you can use with the EPI resource adapter to retrieve and send information to CICS through the EPI. Like the EPI Support classes, it allows you to address fields on a screen. Use the Screen container to get a reference to a field, and then use methods to query and manipulate the field text.

The record is found in the **com.ibm.connector2.cics** package. It is an implementation of the screenable interface, which transfers information between the EPI resource adapter and the record.

The EPIScreenRecord: When you create an EPIScreenRecord:

```
EPIScreenRecord screen = new EPIScreenRecordImpl();
```

you instantiate an **EPIScreenRecordImpl**.

You start a new transaction by passing this record, for example:

```
EPIInteractionSpec epiSpec = new EPIInteractionSpec();
epiSpec.setFunctionName("CESN");
epiSpec.setAID(AIDKey.enter);
epiSpec.setInteractionVerb(EPIInteractionSpec.SYNC_SEND_RECEIVE);
// epiInter is an interaction created elsewhere
epiInter.execute(epiSpec, null, screen);
```

Note the use of *null* as the input record.

The screen information is in the screen object. Other screen information, such as cursor position, is returned to your defined EPIInteractionSpec object. You can then request a specific field by index number, which is a number in the range from 1 to the total number of fields on the screen, or you can use an iterator to request all the fields. The fields are indexed in order starting from the top left of the screen proceeding from left to right to the bottom right of the screen. The iterator returns each field in ascending index order.

So for example you can obtain a field using the index number by coding:

```
EPIFieldRecord field = screen.getField(7);
```

To use the iterator, code the following:

```
java.util.Iterator it = screen.getFields();
while (it.hasNext()) {
    EPIFieldRecord field = (EPIFieldRecord)it.next();
    ....
    ....
}
```

The following is an example of a function that takes a screen record and prints out the screen in a layout suitable for a terminal:

```
public void printScreen(EPIScreenRecord inscr) {
   int col = 1;
   int row = 1;

   System.out.println("————————————————————————————————————————");

    for (int i = 1; i <= inscr.getFieldCount(); i++) {
        try {
            EPIFieldRecord f = inscr.getField(i);
            while (f.getTextRow() > row) {
                System.out.print("\n");
                row++;
                col = 1;
            }
            while (f.getTextCol() > col) {
                System.out.print(" ");
                col++;
            }
            if (f.isDisplay()) {
                System.out.print(f.getText());
                col += f.getText().length();
            }
        }
        catch (ScreenException se) {
        }
    }
    System.out.print("\n");

   System.out.println("————————————————————————————————————————");
}
```

After you have accessed and updated the fields, pass the record back as the input record. If you wish, you can use it again as the output record. For example:

```
epiSpec.setAID(AIDKey.enter);
epiInter.execute(epiSpec, screen, screen);
```

The EPIFieldRecord: Access EPIFieldRecords from an EPIScreenRecord instance rather than creating them directly. The EPIFieldRecord has methods to access the attributes of a field, for example whether it is protected or which colors are available. You can also retrieve and modify text. See *CICS Transaction Gateway: Programming Reference*, for more information about these interfaces in the **com.ibm.connector2.cics** package. The EPIFieldRecord contains the static final variables that define names for color attributes, highlighting and transparency.

The ScreenException: An EPIScreenRecord and EPIFieldRecord can throw exceptions. They are checked exceptions, inherited from the base class ScreenException. See *CICS Transaction Gateway: Programming Reference*, for more information on these exceptions.

Stream Format

The stream is a byte representation of the screen. The number of bytes that are sent to the application, and received from the application, should be the same as the number of bytes on the screen. That is, the number of bytes should equal the product of *screen depth* and *screen width*. For example, if the terminal to which you are connected has a 24 by 80 character screen, the number of bytes that should be flowed to and from the resource adapter is: `24x80 = 1920` bytes.

When providing an input record, you must flow the exact number of bytes on the stream, otherwise the record will be rejected. The byte stream must represent exactly what the screen looks like as seen by the resource adapter. If it does not the record will be rejected.

For each field on the screen, there is a byte preceding the field that represents the attribute byte on a 3270 terminal. On a 3270 screen this byte is displayed as a blank. However, in the byte stream it can contain information about the field. You can select what is placed in this field by specifying an appropriate value in the `EPIInteractionSpec` `setOutputAttributeType` method. For example, this byte could contain a blank, which is the base attribute, or it could contain a value which represents the color attribute for that field.

A special option is `EPIInteractionSpec.ATTRIBUTE_MARKER`. This stores the value `EPIInteractionSpec.MARKER_BYTE` in that location. This enables a record to locate a field dynamically, without needing prior knowledge of the screen format, for example a BMS map.

Writing LogonLogoff classes

LogonLogoff classes are specified at deployment and used to logon to sign-on capable terminals, or to terminals that install as *sign-on unknown*.

It is recommended that you use sign-on incapable terminals, in which case you do not need the LogonLogoff classes.

If you choose to use the classes, implement the `com.ibm.connector2.cci.LogonLogoff` interface which has the following interface definition:

```
public interface LogonLogoff {
  public void logoff(javax.resource.cci.Connection conn);
  public void logon(javax.resource.cci.Connection conn,
                    javax.security.auth.Subject security);
}
```

This class is only required for the EPI resource adapter. You do not need to implement the logoff method because this is never called. However, you must provide a dummy implementation so that the class can be compiled. You are passed a connection and a security subject with the logon method signature. The logon is driven in the same way as for applications that communicate with CICS using the EPI resource adapter. You create interactions using this connection and, when finished, you close the interaction. For example:

```
Interaction epiInt = (Interaction)(conn.createInteraction());
EPIInteractionSpec spec = new EPIInteractionSpec();

//-----------------------------------------------------------------
// configure the spec to perform a CESN, and execute the call
//-----------------------------------------------------------------
spec.setAID(AIDKey.enter);
spec.setFunctionName("CESN");
spec.setInteractionVerb(EPIInteractionSpec.SYNC_SEND_RECEIVE);
EPIScreenRecord screen = new EPIScreenRecordImpl();
epiInt.execute(spec,null,screen);
```

Close the interaction when you have finished with it. For example:

```
epiInt.close();
```

Note: Do not close the connection within the LogonLogoff class.

The credentials with which you logon are held as *Subject object*. To retrieve this information you need to get an iterator from the private credentials. There is a single entry within the private credentials of type *PasswordCredential*. You can obtain the userid and password from this entry as follows:

```
Iterator it = security.getPrivateCredentials().iterator();
PasswordCredential pc = null;
if (it.hasNext()) {
  pc = (PasswordCredential)it.next();
}
if (pc == null) {
  throw new javax.resource.spi.SecurityException("
    Unable to logon, No Security Information Provided");
}
String user = pc.getUserName();
String pass = new String(pc.getPassword());
```

If there are any problems, throw a `javax.resource.spi.SecurityException`.

Java security

You might need to grant your LogonLogoff class the Java security permission, to enable it to retrieve the credential information from the subject passed to it:

```
permission javax.security.auth.PrivateCredentialPermission
"javax.resource.spi.security.PasswordCredential * \"*\"", "read";
```

Samples

JCA EPI sample programs are provided in the samples subdirectory of your CICS Transaction Gateway installation or as a deployable EAR in the <install_path> deployable subdirectory. These are documented in “Resource adapter samples” on page 102.

Using the J2EE CICS resource adapters in a nonmanaged environment

You can use the resource adapters in a nonmanaged environment. In this environment, you are responsible for:

- Defining the EIS connection
- Creating the ConnectionFactory object
- Providing your own connection pooling
- Supplying your log writer
- Managing transactions

Your nonmanaged environment can be either inside, or outside, a J2EE server environment. The resource adapters provide a default connection manager to support execution within the nonmanaged environment.

Transaction management applies only to the ECI resource adapter. See “Transaction management” on page 89 for information on managing transactions in a nonmanaged environment.

Creating the appropriate ConnectionFactory object

Your application needs to get an appropriate ConnectionFactory object. In the managed environment, the server or application does this for you, and you can reference it using JNDI (see “Storing ConnectionFactory objects” on page 99). In the nonmanaged environment, unless you have previously registered one that you can access, you must create a ConnectionFactory object with the appropriate EIS connection information.

Creating an ECI ConnectionFactory

You must first create an ECIManagedConnectionFactory and set the appropriate properties on this object. The properties are the same as the deployment parameters described in *J2EE setup and configuration*, in the *CICS Transaction Gateway: Administration* book for your operating system. These are accessible using setter and getter methods. The *J2EE Programming Reference* documentation lists the setter and getter methods for the ECIManagedConnectionFactory and shows the relationship between deployment parameters and properties. The following example shows how to create a ConnectionFactory for ECI:

```
ECIManagedConnectionFactory eciMgdCf = new ECIManagedConnectionFactory();
eciMgdCf.setConnectionURL("local:");
eciMgdCf.setPortNumber(new Integer(0));
eciMgdCf.setServerName("tp600");
eciMgdCf.setLogWriter(new java.io.PrintWriter(System.err));
eciMgdCf.setUserName("myUser");
eciMgdCf.setPassword("myPass");
eciMgdCf.setTraceLevel(new
                  Integer(ECIManagedConnectionFactory.RAS_TRACE_ENTRY_EXIT));
ConnectionFactory cxf = (ConnectionFactory)eciMgdCf.createConnectionFactory();
```

Creating an EPI ConnectionFactory

You must first create an EPIManagedConnectionFactory and set the appropriate properties on this object. The properties are the same as the deployment parameters described in *J2EE setup and configuration*, in the *CICS Transaction Gateway: Administration* book for your operating system. This process is similar to that for creating an ECI ConnectionFactory. The following example shows how to create a ConnectionFactory for EPI:

```
EPIManagedConnectionFactory epiMgdCf = new EPIManagedConnectionFactory();
epiMgdCf.setConnectionURL("local:");
epiMgdCf.setPortNumber(new Integer(0));
epiMgdCf.setServerName("tp600");
epiMgdCf.setLogWriter(new java.io.PrintWriter(System.err));
epiMgdCf.setUserName("myUser");
epiMgdCf.setPassword("myPass");
epiMgdCf.setSignonType(new Integer(0)); // sign-on capable terminal
epiMgdCf.setLogonLogoffClass("com.acme.companyApp.ourCICSLogon");
epiMgdCf.setTraceLevel(new
             Integer(EPIManagedConnectionFactory.RAS_TRACE_ERROR_EXCEPTION));
ConnectionFactory cxf = (ConnectionFactory)epiMgdCf.createConnectionFactory();
```

Storing ConnectionFactory objects

You can store ConnectionFactory objects for later reuse, so that your application doesn't need to rebuild them. Inside a J2EE server environment, IBM recommends that you register your ConnectionFactory object, which has links to your EIS connection information, in the J2EE *Java Naming and*

Directory Interface (JNDI) service. This makes migration from nonmanaged to managed Java environments easier because applications can get ConnectionFactory objects in the same manner. However, this may not be possible outside a JNDI environment unless either an *LDAP* server, or an appropriate *JNDI Service Provider*, is available within your environment.

The resource adapter ConnectionFactory objects support both the *serializable* and *referenceable* Java interfaces. This means that you can choose how to register them in the JNDI. For more information, refer to the *J2EE Connector Architecture Specification*.

If you plan to use serializable interfaces, refer to *J2EE Tracing*, in the *CICS Transaction Gateway: Administration* book for your operating system. This gives information on how serialization and deserialization of ConnectionFactory objects affects the setting of the LogWriter property.

Running the J2EE CICS resource adapters in a nonmanaged environment

In a J2EE environment, all of the required Java libraries should be available. You might need to ensure that your J2EE server adds the following delivered jar files to your class path. These files are in the <install_path>\classes subdirectory:

- cicsj2ee.jar
- ctgclient.jar
- ctgserver.jar (required only for local: protocol)
- ccf2.jar
- connector.jar
- screenable.jar (required for the EPIScreenRecord)

Outside a J2EE enviroment, you must ensure that, as well as the above libraries being listed in the class path, the following Java extensions are also available:

- JAAS (required for EPI resource adapter)
- JTA (required for the ECI resource adapter)

JAAS is included with IBM JREs and JDKs by default. The JTA library is available from the Sun Java website.

All of the above libraries and extensions should be available from the J2EE server libraries.

Compiling applications

To compile supplied applications in both managed and nonmanaged environments, include the following in the CLASSPATH:

- cicsj2ee.jar (required for access to Connection and Interaction Specs)
- ctgclient.jar (required for AIDkey objects)
- ccf2.jar (required for creating LogonLogoff classes)
- connector.jar (required for all resource adapter applications)
- screenable.jar (required if using the EPI Screen Record)

Compiling and running J2EE components

If you develop a J2EE component that passes back the EPI Screen Record as a return parameter, your deployment tool needs the following jar files:

- cicsj2ee.jar
- screenable.jar

An EJB client that receives an EPI Screen Record also needs these jar files on the class path.

Security credentials and the CICS resource adapters

Security Credentials for accessing CICS can come from three different places. These are the ConnectionSpec properties, the deployed security credentials, or the server itself (for nonmanaged environments, the third option does not apply). The precedence for these credentials is:

1. The Server Supplied Credentials (highest precedence)
2. The ConnectionSpec Supplied Credentials
3. The Deployed Security Credentials.

Managed enterprise applications can be deployed with ″container″ or ″application″ as a security choice. If ″container″ is specified, the J2EE application server will provide the credentials by means of a user interface. If ″application″ is specified, security is determined from the deployment properties and can be overridden by the ConnectionSpec.

J2EE tracing

In a nonmanaged environment where the DefaultConnectionManager is used the application can set the LogWriter property on the class to define where trace messages are sent. It is important to note however that in a nonmanaged environment, if the ConnectionFactory is serialized for storage the `LogWriter` must be set after deserialization in order for it to be used, as it is not restored automatically after deserialization. This process is shown in the following example:

```
ECIManagedConnectionFactory MCF = new ECIManagedConnectionFactory();
MCF.setLogWriter(myLogWriter);

ECIConnectionFactory cf = MCF.createConnectionFactory();
objOutStream.write(cf);

ECIConnectionFactory cf2 = (ECIConnectionFactory) objInStream.read();
DefaultConnectionManager.setLogWriter(myLogWriter);
```

Issues with tracing if ConnectionFactory serialized

As described above, if you use the serializable interface to store your ConnectionFactory then you lose the reference to your LogWriter. This is because LogWriters are not serializable and cannot be stored. When you deserialize your ConnectionFactory it will not contain a reference to the LogWriter. To ensure that your LogWriters are stored on any connections created from this ConnectionFactory you must do the following. This only applies in a nonmanaged environment.

```
DefaultConnectionManager.setLogWriter(new java.io.PrintWriter(System.err));
Connection Conn = (Connection)cxf.getConnection();
```

The `setLogWriter` method on the `DefaultConnectionManager`, which is supplied with the resource adapters, is a static method. The example above shows how to set the log to output the `System.err`. The trace level applied to the `ManagedConnectionFactory` remains.

Resource adapter samples

The samples consist of an ECI COMMAREA sample and an EPI sample. They show how to use the CICS resource adapters as well as demonstrating how to write custom records that implement the javax.resource.cci.Streamable interface. For information on how to deploy the ECI and EPI resource adapters, see *J2EE setup and configuration*, in the *CICS Transaction Gateway: Administration* guide for your operating system.

ECI COMMAREA sample

| The ECI COMMAREA sample consists of a stateless session bean, a client application, and a custom record that demonstrates using the Streamable interface. The following files are part of the sample:

ECIDateTime.java
Enterprise bean remote interface

ECIDateTimeHome.java
Enterprise bean home interface

ECIDateTimeBean.java
Enterprise bean implementation

ECIDateTimeClient.java
Enterprise bean client program

JavaStringRecord.java
Custom Record

Ejb-jar-eci-1.1.xml
Example of a deployment descriptor

The deployment descriptor is an example of an EJB 1.1–compliant deployment descriptor for this Enterprise Bean. If you wish to package it up into a jar file, rename it to `Ejb-jar.xml` and store it in the META-INF directory of the jar file. It may require further entries if it is to be deployed into an EJB 2.0–compliant environment.

See your J2EE Server documentation for information on how to compile and deploy the bean within your environment. However, you need to ensure that the following jar files are also available on the CLASSPATH:

- cicsj2ee.jar
- connector.jar
- ctgclient.jar
- ccf2.jar

The enterprise bean looks for an ECI connection factory named `java:comp/env/ECI`. The bean must refer to this resource when deployed. Refer to your J2EE Server documentation on how to deploy the resource adapter with an entry in the JNDI with this name. The client program looks for the ECIDateTime bean with a name of `ECIDateTimeBean1`. See your J2EE Server document for details of how to setup the bean with this JNDI name.

You will need to install the sever sample program EC01 on your CICS Server. This file can be found in the samples\server subdirectory of your CICS Transaction Gateway installation. Further details of this sample can be found in the samples.txt file in the samples folder.

The bean is a simple bean that outputs the date and time as known to the CICS Server, and can be deployed as a bean-managed transaction. The Custom record takes a COMMAREA and converts it to a string. Ensure that the EC01 sample program, which you installed on your CICS server, sends its results in ASCII, as the COMMAREA is expected in ASCII. The JavaStringRecord does however allow for the selection of other encodings, and is commented using JavaDoc. The Client program takes no parameters. If your CICS server is running on z/OS, the EC01 sample program will return its results in EBCDIC rather than ASCII. To resolve this, update the DFHCNV table by adding lines similar to the following:

```
*
* CTG Sample conversion
*
*
         DFHCNV TYPE=ENTRY,RTYPE=PC,RNAME=EC01,USREXIT=NO,             *
               SRVERCP=037,CLINTCP=8859-1
         DFHCNV TYPE=SELECT,OPTION=DEFAULT
         DFHCNV TYPE=FIELD,OFFSET=0,DATATYP=CHARACTER,DATALEN=18,      *
               LAST=YES
```

EPI sample

The EPI Sample consists of a stateful session bean, a client application, a custom record which demonstrates the use of the Screenable interface, and a custom LogonLogoff class.The following files are part of the EPI Sample:

EPIPlayScript.java
Enterprise bean remote interface

EPIPlayScriptHome.java
Enterprise bean home interface

EPIPlayScriptBean.java
Enterprise bean implementation

EPIPlayScriptClient.java
Enterprise bean client program

CICSCESNLogon.java
A LogonLogoff class

Ejb-jar-epi-1.1.xml
Example of a deployment descriptor

The deployment descriptor is an example of an EJB 1.1-compliant deployment descriptor for this Enterprise Bean. If you wish to package it up into a jar file, rename it to `Ejb-jar.xml` and store it in the META-INF directory of the jar file. It may require further entries if it is to be deployed into an EJB 2.0-compliant environment.

Your J2EE Server documentation describes how to compile and deploy the bean within your environment. However, you need to ensure that the following jar files are also available on the CLASSPATH:

- cicsj2ee.jar
- connector.jar
- ctgclient.jar
- ccf2.jar
- screenable.jar

The Enterprise Bean looks for an EPI connection factory named `java:comp/env/EPI`. See your J2EE Server's documentation for details of how deploy the resource adapter under this reference in the JNDI. When deploying the bean into your environment you need to supply this reference for the bean to find the resource. The client program looks for the EPIPlayScript bean with a name of `EPIPlayScript1`. Refer to your J2EE Server documentation for details of how to setup the bean with this name in the JNDI namespace. The bean can be deployed as a bean-managed transaction.

The bean is designed to take a series of commands and drive a 3270 interaction. Once the commands are complete, the field text is returned as a string array based on fields requested to be returned by the script. The client can then look at these field texts and send more commands to drive that interaction if necessary. The commands that drive the 3270 screen are as follows:

S(txn) Start transaction "txn"

F(x)="Text"
Set field number x to "Text". Field numbers start at 1.

P(aid) Press key 'aid'

C(row, col)
place cursor at row, col (row and col start at 1)

R(x) Adds the text of the field at the given field number to the string array that will be returned. Field numbers start at 1.

So an example of a script might be:

```
S(CESN)F(7)="myuser"F(10)="mypass"P(enter)R(1)
```

The EPIPlayScriptClient program takes no parameters; it has a default command sequence coded into it. Experiment by changing this command sequence or enhancing the sample.

The `CICSCESNLogon.java` sample contains example code on how to logon to a CICS Transaction Server for z/OS system. The code is designed to work for English systems and might have to be tailored for other versions of CICS and languages. In order to use this class, deploy it as part of the sample bean and reference it when you deploy the EPI resource adapter. For more information about how to deploy the EPI resource adapter see *J2EE setup and configuration*, in the *CICS Transaction Gateway: Administration* book for your operating system.

Assistance in coding CCI applications

Connector specification API Javadoc

You can obtain the connector architecture API Javadoc from the Sun Web site, this will assist in the coding of your CCI applications and provides information such as the exceptions used by CCI implementations.

J2EE Connector Specification API

IBM recommends that you get the *J2EE Connector Specification* document from Sun's Web site at java.sun.com/j2ee/download.html, to help in coding your CCI applications. It contains information such as the exceptions used in CCI applications.

Chapter 8. Programming in C and COBOL

This information contains information about the external access interfaces specific to C and COBOL. It does not deal with testing or debugging ECI, EPI, and ESI applications; refer instead to the programming documentation for the environment in which you are working.

Overview of the programming interface for C and COBOL

The interfaces provided for the C and COBOL programming languages are similar. Parameter blocks are used to pass data between the Client application and the ECI, EPI and ESI.

A user application must be constructed as a single process, though in environments in which a process can generate several threads, the user application can be multi-threaded.

Note that the COBOL programming language is only available for Windows.

Because the COBOL and C field names are similar, most of the examples in this chapter only use the C names. Refer to Table 14, Table 16 on page 111, and Table 18 on page 119 for a comparison.

Making External Call Interface calls from C and COBOL programs

This section describes how to make ECI calls to a CICS server from a COBOL or C Client application. Table 14 shows the field names in C and COBOL data structures that correspond to the ECI terms described in "Input and output information for external calls to CICS" on page 9

Table 14. ECI terms and corresponding fields in C and COBOL

ECI term	C structure.field	COBOL structure.field
Abend code	ECI_PARMS.eci_abend_Code	ECI-PARMS.ECI-ABEND-CODE
COMMAREA	ECI_PARMS.eci_commarea	ECI-PARMS.ECI-COMMAREA
ECI timeout	ECI_PARMS.eci_timeout	ECI-PARMS.ECI-TIMEOUT
LUW control	ECI_PARMS.eci_extend_mode	ECI-PARMS.ECI-EXTENDED
LUW identifier	ECI_PARMS.eci_luw_token	ECI-PARMS.ECI-LUW-TOKEN
Password	ECI_PARMS.eci_password ECI_PARMS.eci_password2	ECI-PARMS.ECI-PASSWORD ECI-PARMS.ECI-PASSWORD2
Program name	ECI_PARMS.eci_program_name	ECI-PARMS.ECI-PROGRAM-NAME

Table 14. ECI terms and corresponding fields in C and COBOL (continued)

ECI term	C structure.field	COBOL structure.field
Server name	ECI_PARMS.eci_system_name	ECI-PARMS.ECI-SYSTEM-NAME
TPNName	ECI_PARMS.eci_tpn	ECI-PARMS.ECI-TPN
TranName	ECI_PARMS.eci_transid	ECI-PARMS.ECI-TRANSID
User ID	ECI_PARMS.eci_userid	ECI-PARMS.ECI-USERID

CICS_ExternalCall

Use CICS_ExternalCall for making program link calls, status information calls, and reply solicitation calls. Use the ECI parameter block (ECI_PARMS for C and ECI-PARMS for COBOL) for passing parameters to the ECI. The eci_call_type parameter in the ECI parameter block indicates the type of CICS_ExternalCall. The following examples show the format of the request and associated declarations:

For C programs:

```
ECI_PARMS      EciBlock;
cics_sshort_t  Response;
.
.
.
Response = CICS_ExternalCall(&EciBlock);
```

For COBOL programs:

```
CALL CICSEXTERNALCALL
USING BY REFERENCE ECI-PARMS
RETURNING ECI_ERROR_ID.
```

Program link calls

Fill in the required fields in the ECI parameter block. Pass any data required by the program you are linking to in the COMMAREA.

Use eci_call_type to define an ECI request as either synchronous or asynchronous:

- ECI_SYNC for a synchronous program link call
- ECI_ASYNC for an asynchronous program link call

Managing logical units of work

To start a logical unit of work, set the eci_extend_mode parameter to ECI_EXTENDED and the eci_luw_token parameter to zero, when making a program link call. The Client daemon generates a LUW identifier which is returned in the eci_luw_token field. This identifier must be input to all subsequent calls for the same unit of work. To call the last program in a LUW, set the eci_extend_mode parameter to ECI_NO_EXTEND. To end a LUW without linking to a program, set the eci_extend_mode parameter to ECI_COMMIT or ECI_BACKOUT to commit or back out changes to recoverable resources.

Table 15 shows how you can use combinations of eci_extend_mode, eci_program_name, and eci_luw_token parameter values to perform tasks associated with managing logical units of work through ECI. In each case you must also store appropriate values in other fields for the call type you have chosen.

Table 15. Logical units of work in ECI

Task to perform	**Parameters to use**
Call a program that is to be the only program of a logical unit of work. One request flows from client to server and a reply is sent to the client only after all the changes made by the specified program have been committed.	Set up the parameters as follows: • **eci_extend_mode**: ECI_NO_EXTEND • **eci_program_name**: provide it • **eci_luw_token**: zero
Call a program that is to start an extended logical unit of work.	Set up the parameters as follows: • **eci_extend_mode**: ECI_EXTENDED • **eci_program_name**: provide it • **eci_luw_token**: zero Afterwards, save the token from **eci_luw_token**.
Call a program that is to continue an existing logical unit of work.	Set up the parameters as follows: • **eci_extend_mode**: ECI_EXTENDED • **eci_program_name**: provide it • **eci_luw_token**: provide it
Call a program that is to be the last program of an existing logical unit of work, and commit the changes.	Set up the parameters as follows: • **eci_extend_mode**: ECI_NO_EXTEND • **eci_program_name**: provide it • **eci_luw_token**: provide it
End an existing logical unit of work, without calling another program, and commit changes to recoverable resources.	Set up the parameters as follows: • **eci_extend_mode**: ECI_COMMIT • **eci_program_name**: null • **eci_luw_token**: provide it
End an existing logical unit of work, without calling another program, and back out changes to recoverable resources.	Set up the parameters as follows: • **eci_extend_mode**: ECI_BACKOUT • **eci_program_name**: null • **eci_luw_token**: provide it

If an error occurs in one of the calls of an extended logical unit of work, you can use the eci_luw_token field to see if the changes made so far have been backed out, or are still pending. See the description of the eci_luw_token field in *CICS Transaction Gateway: Programming Reference* for more information. If the changes are still pending, end the logical unit of work with another program link call, either committing or backing out the changes.

ECI timeouts

Use the **eci_timeout** field in the ECI parameter block to specify the timeout value. If a timeout occurs either the ECI_ERR_RESPONSE_TIMEOUT code or the ECI_ERR_REQUEST_TIMEOUT code is returned.

See "Timeout of the ECI request" on page 14 for more information on ECI timeouts.

Reply solicitation calls

Use one of the following call types to solicit replies for an asynchronous program link call. Unique message qualifiers for specific replies must be created by the Client application.

ECI_GET_REPLY
For a reply solicitation call that gets any outstanding reply for any asynchronous call, if any reply is available.

ECI_GET_REPLY_WAIT
For a reply solicitation call that gets any outstanding reply for any asynchronous call, waiting if no replies are available.

ECI_GET_SPECIFIC_REPLY
For a reply solicitation call that gets any outstanding reply for a given asynchronous call, if any reply is available.

ECI_GET_SPECIFIC_REPLY_WAIT
For a reply solicitation call that gets any outstanding reply for a given asynchronous call, waiting if no replies are available.

Security in the ECI

The Client application can specify the user ID and password by setting eci_userid and eci_password or eci_userid2 and eci_password2 in the ECI parameter block. Use eci_userid and eci_password if the user ID and password names are 8 characters or less in length, or eci_userid2 and eci_password2 if the names can be more than 8 characters in length.

You can set a default user ID and password for the connection. See "Making External Security Interface calls from C and COBOL programs" on page 119 for more information.

Making External Presentation Interface calls from C and COBOL programs

This section describes how to run a 3270-based program on a CICS server using EPI calls from a C or COBOL application. Table 16 shows the field names in C and COBOL data structures that correspond to the terminal attributes described in "Terminal characteristics" on page 19.

Table 16. C and COBOL field names corresponding to terminal attributes

EPI term	C structure.field	COBOL structure.field
Code page	CICS_EpiAttributes_t.CCSId	CICS-EPIATTRIBUTES.CCSID
Color	CICS_EpiDetails_t.Color	CICS-EPIDETAILS.COLOR
Columns	CICS_EpiDetails_t.NumColumns	CICS-EPIDETAILS.NUMCOLUMNS
Device type	CICS_EpiAddTerminal(,,,DevType,,,,)	CICSEPIADDTERMINAL.(,,,DEVTYPE,,,,)
Error last line	CICS_EpiDetails_t.ErrLastLine	CICS-EPIDETAILS.ERRLASTLINE
Error message color	CICS_EpiDetails_t.ErrColor	CICS-EPIDETAILS.ERRCOLOR
Error message highlight	CICS_EpiDetails_t.ErrHilight	CICS-EPIDETAILS.ERRHILIGHT
Error message intensity	CICS_EpiDetails_t.ErrIntensity	CICS-EPIDETAILS.ERRINTENSITY
Extended highlight	CICS_EpiDetails_t.Hilight	CICS-EPIDETAILS.HILIGHT
Install timeout	CICS_EpiAttributes_t.InstallTimeOut	CICS-EPIATTRIBUTES.INSTALLTIMEOUT
Map name	CICS_EpiEventData_t.MapName	CICS-EPIEVENTDATA.MAPNAME
Map set name	CICS_EpiEventData_t.MapSetName	CICS-EPIEVENTDATA.MAPSETNAME
Maximum data	CICS_EpiDetails_t.MaxData	CICS-EPIDETAILS.MAXDATA
Netname	CICS_EpiDetails_t.NetName	CICS-EPIDETAILS.NETNAME
Password	CICS_EpiAttributes_t.Password	CICS-EPIATTRIBUTES.EPI-PASSWORD
Read timeout	CICS_EpiAttributes_t.ReadTimeOut	CICS-EPIATTRIBUTES.READTIMEOUT
Rows	CICS_EpiDetails_t.NumLines	CICS-EPIDETAILS.NUMLINES
Server name	CICS_EpiDetails_t.System	CICS-EPIDETAILS.SYSTEM
Sign-on capability	CICS_EpiAttributes_t.SignonCapability	CICS-EPIATTRIBUTES.SIGNONCAP
Terminal ID	CICS_EpiDetails_t.Termid	CICS-EPIDETAILS.TERMID
Userid	CICS_EpiAttributes_t.Userid	CICS-EPIATTRIBUTES.EPI-USERID

EPI versions

Only version 2 of the EPI is supported for new applications. Existing applications that use EPI version 1 are supported for compatibility with earlier versions.

EPI Initialization and termination

Any application that needs to use EPI must call the CICS_EpiInitialize function to initialize EPI. Until this call is made, no other EPI function is allowed. The CICS_EpiInitialize function takes a parameter indicating the version of the EPI for which the application was coded. This is to ensure that existing applications continue to run without change if the EPI is extended.

Before an EPI application ends, it must call the CICS_EpiTerminate function to terminate EPI cleanly.

If the Client Daemon is restarted while an application is active, the application must reissue CICS_EpiInitialize and reinstall all the terminals. Restarting the Client Daemon while an application is active is not recommended.

Adding a terminal to CICS

Use the CICS_EpiAddTerminal function or the CICS_EpiAddExTerminal function to add terminals to CICS.

Terminal indexes

The CICS_EpiAddTerminal and CICS_EpiAddExTerminal functions return a *terminal index*, which must be passed on subsequent EPI function calls to indicate the terminal to which the function is to apply. Each index identifies a combination of server name and terminal ID. The terminal index supplied is the first available integer starting from 0.

Terminal indexes are unique within a Client application, but not across applications, so each application gets terminal index zero for the first terminal it installs, and so on.

When the terminal has been deleted, the terminal index value becomes free and can be reused when another terminal is added. The server deletes the terminal if it was autoinstalled.

Install timeout

The length of time that an application will wait for a terminal to be installed is specified in the InstallTimeOut field in the CICS_EpiAttributes_t structure passed to the **CICS_EpiAddExTerminal** function.

If no response is received from the server within the specified interval, control is returned to the invoking application with the return code set to CICS_EPI_ERR_RESPONSE_TIMEOUT.

Deleting a terminal

If a terminal is no longer required, it can be deleted by invoking either the **CICS_EpiDelTerminal** or **CICS_EpiPurgeTerminal** function.

Use the **CICS_EpiDelTerminal** if no transaction is running against the terminal and there are no unprocessed events outstanding.

Use the **CICS_EpiPurgeTerminal** function if the terminal is to be deleted without regard to any transaction that may be running against the terminal or unprocessed events for that terminal.

Starting transactions

To start a transaction, call the **CICS_EpiStartTran** function. There are two ways of specifying the transaction to be started and the data to be associated with it:

1. Supply the transaction identifier as a parameter to the call (**TransId**), and supply any transaction data in the **Data** parameter.
2. Combine a transaction identifier and transaction data into a 3270 data stream, and supply the data stream as a parameter to the call (**Data**).

The server might have to:

- Authenticate the userid and password for the terminal ″operator″.
- Grant authority, based on the authenticated userid, to access the resources required for the execution of each transaction.

The frequency with which the userid and password are authenticated by the server depends on whether the terminal has been defined as sign-on capable or sign-on incapable; see "Security in the EPI" on page 21.

Sending and receiving data

When a transaction sends data to a terminal, the EPI generates either a CICS_EPI_EVENT_SEND event or a CICS_EPI_EVENT_CONVERSE event.

The CICS_EPI_EVENT_SEND event indicates that data was sent but that no reply is required. Typically this would result from an EXEC CICS SEND command, but in some servers it would result from an EXEC CICS CONVERSE command. (In the latter case, a CICS_EPI_EVENT_CONVERSE event occurs later to tell the application to send a data stream back to the transaction in the server.)

The CICS_EPI_EVENT_CONVERSE event indicates that a reply is required, and would typically result from an EXEC CICS RECEIVE or EXEC CICS CONVERSE command. The application must respond to this event by issuing a **CICS_EpiReply** call to provide the response data. The **CICS_EpiReply** function should be issued only to respond to a CICS_EPI_EVENT_CONVERSE event; if it is issued at any other time, an error is returned.

Managing pseudoconversations

The CICS_EPI_EVENT_END_TRAN event tells the application whether the transaction just ended has specified a transaction to process the next input, and which transaction has been specified. The application must not attempt to start a different transaction, but must use **CICS_EpiStartTran** to start the transaction specified by the CICS_EPI_EVENT_END_TRAN event.

Events and callbacks

Use the **CICS_EpiGetEvent** function to collect events.The EPI puts information in a *CICS_EpiEventData_t* structure to indicate the event that occurred and any associated data. It also indicates whether there are more events still waiting in the queue.

The application can synchronize the processing of these events with its other activities in one of three ways:

- Polling.
- Blocking.
- Callback notification. *Callback* is a way for another thread to notify your application thread that an event has happened.

Polling

The **CICS_EpiGetEvent** call can be made in a polling mode by specifying CICS_EPI_NOWAIT for the **Wait** parameter. If no event is waiting to be collected, the function returns immediately with an error code. This is the mechanism that you would have to adopt in a single-user single-threaded environment, where the application might alternately poll the keyboard for user activity and poll the EPI for event activity. This mechanism is not recommended.

Blocking

The **CICS_EpiGetEvent** call can be made in a blocking mode by specifying CICS_EPI_WAIT for the **Wait** parameter. If no event is waiting to be collected, the function waits and does not return until an event becomes available. You could use this mechanism in a multithreaded environment, where a secondary thread could be dedicated to event processing. It could also be used after a notification by callback, because the event information is known to be available.

Callback notification

Callback routines can be used in C but are not available in Cobol.

When you define a terminal, you can use the optional parameter **NotifyFn** to provide the address of a callback routine that the EPI is to call whenever an event occurs against that terminal.

Note: Some compilers do not support the use of callback routines. Consult your compiler documentation for more information.

An application should carry out the minimum of processing in its callback routine, and never block in the specified routine before returning to the EPI. The routine itself cannot make EPI calls. You decide what it should do when the notification is received. For example, in a multithreaded environment, it might post a semaphore to signal another thread that an event has occurred. In a Windows environment, it might post a message to a window to indicate to the window procedure that an event has occurred. Other actions will be appropriate for other environments.

When the callback routine is called, it is passed a single parameter—the terminal index of the terminal against which the event occurred. This allows the same callback routine to be used for more than one terminal.

Processing events

The **CICS_EpiGetEvent** function returns information about an event in the *CICS_EpiEventData_t* structure. The **Event** field in this structure contains the name of the event:

- CICS_EPI_EVENT_SEND
- CICS_EPI_EVENT_CONVERSE
- CICS_EPI_EVENT_END_TRAN
- CICS_EPI_EVENT_START_ATI
- CICS_EPI_EVENT_ADD_TERM
- CICS_EPI_EVENT_END_TERM

The application should process events as quickly as possible.

When a Client application is driven with an event or callback, it must issue a **CICS_EpiGetEvent** to get the associated event. In certain timing conditions, the CICS_EPI_EVENT_START_ATI may already have been notified from a previous **CICS_EpiGetEvent**. The **CICS_EpiGetEvent** issued after the callback can receive CICS_EPI_ERR_NO_EVENT (if CICS_EPI_NOWAIT is specified for the **Wait** parameter) or wait until a subsequent event is received (if CICS_EPI_WAIT is specified for the **Wait** parameter). Note that this can happen after a CICS_EPI_EVENT_START_ATI is received.

Automatic transaction initiation (ATI)

The CICS server API call EXEC CICS START allows a server program to start a transaction on a particular terminal. This mechanism, called *Automatic*

Transaction Initiation (ATI), requires additional programming at the client side to handle the interaction between these transactions and normal client-initiated transactions.

ATIs are queued for a terminal while a transaction is in progress. By default ATI requests are held, and not started against a terminal. The CICS_EPIATIState function enables and disables ATI requests. If ATIs are enabled, they are run only when the terminal is in an idle state (no transaction is currently running against the terminal). The ATI is started when the CICS_EPI_EVENT_START_ATI event is retrieved.

3270 data streams for the EPI

The supplied C header file, cics3270.h, and the COBOL copybook cics3270.cbl, contain constants and conversion tables that you will find useful in handling 3270 data streams.

EPI to CICS (Inbound data streams)

EPI applications send 3270 data to CICS on calls to the following functions:

- **CICS_EpiStartTran**
- **CICS_EpiReply**.

The format in both cases is the same. The data stream must be a minimum of 3 non-null bytes, representing the AID and cursor address; the sole exception to this is if the AID represents the CLEAR key or a PA key, when the data stream may consist of the AID only. These fields are passed to the CICS transaction in the EIBAID and EIBCPOSN fields of the EIB.

AID (1 byte)	Cursor address (2 bytes)	Data buffer (variable length)

The contents of the data buffer consist of:

- ASCII displayable characters with embedded 3270 control characters, when it is passed to an EXEC CICS RECEIVE MAP command.
- User-specified data, when it is passed to an EXEC CICS RECEIVE command.

On starting a transaction, the transaction ID is extracted from the start of the data buffer as follows:

- If a set buffer address (SBA) order is present at the start of the data buffer, the transaction ID is extracted from the 4th through 7th bytes of the buffer.
- If an SBA is not present at the start of the data buffer, the transaction ID is extracted from the 1st through 4th bytes of the buffer.

In either case, the transaction ID may be shorter than 4 bytes, being delimited by either another SBA, an ASCII space, or the end of the string.

The contents of the data buffer passed on the start of a CICS transaction are available to the transaction in response to an initial EXEC CICS RECEIVE command.

When the application replies, the contents of the data buffer are available in an unconverted form in response to an EXEC CICS RECEIVE command or converted to a BMS structure in response to an EXEC CICS RECEIVE MAP command.

Note: It is the EPI programmer's responsibility in the latter case to ensure that the data is formatted correctly so that the conversion succeeds.

CICS to EPI (Outbound data streams)

The 3270 commands are either write or read commands, instructing the EPI to process the data or to reply with data respectively.

On a CICS_EPI_EVENT_SEND event, the command is one of the following 3270 write commands:

- Write
- Erase/Write
- Erase/Write Alternate
- Erase All Unprotected.

The first three commands are followed by a write control character (WCC) and data. An Erase All Unprotected command has neither WCC nor data. The Write Structured Field command is not generated by CICS and is therefore not supported for the EPI.

Command (1 byte)	Write control character (1 byte)	Data buffer (variable length)

The contents of the data buffer consist of:

- ASCII displayable characters with embedded 3270 control characters, when it is passed from an EXEC CICS SEND MAP command.
- User-specified data, when it is passed from an EXEC CICS SEND command.

A CICS_EPI_EVENT_CONVERSE event specifies a read command. The contents of the data stream vary with the source of the event, as follows:

- If the event is the result of an EXEC CICS RECEIVE command, the data buffer might contain data sent by the transaction, or it might be empty. The EPI program should reply when the data to be sent is available.
- If the event is the result of an EXEC CICS RECEIVE BUFFER command, the data buffer contains the 3270 Read Buffer command. This should be processed as described in the *3270 Data Stream Programmer's Reference*.

3270 order codes provide additional control function

3270 orders are included in both inbound and outbound data streams to provide additional control function. Table 17 lists the order codes that may occur in 3270 data streams, and shows whether they relate to inbound or outbound data streams, or both.

Table 17. Order codes occurring in 3270 data streams

Order code	Inbound	Outbound
Start field (SF)	Yes	Yes
Start field extended (SFE)	Yes	Yes
Set buffer address (SBA)	Yes	Yes
Set attribute (SA)	Yes	Yes
Modify field (MF)	No	Yes
Insert cursor (IC)	No	Yes
Program tab (TB)	No	Yes
Repeat to address (RA)	No	Yes
Erase unprotected to address (EUA)	No	Yes
Graphic escape (GE)	No	No

Note: The *3270 Data Stream Programmer's Reference* states that the SFE, SA, and MF orders are not supported in ASCII. However, they do occur in 3270 data streams for the EPI, where they take the following values:

```
SFE     X'10'
SA      X'1F'
MF      X'1A'
```

Each of these orders is followed by one or more attribute type-value pairs. The count of attribute pairs and the attribute type are both binary values, and are thus as defined in the *3270 Data Stream Programmer's Reference*. However, the contents of the attribute value field may vary from those defined in the *3270 Data Stream Programmer's Reference* as follows:

- If the attribute type is less than or equal to X'C0' (for example, a color), the attribute value is defined as an EBCDIC value in the *3270*

Data Stream Programmer's Reference. The EPI uses the ASCII equivalent of the EBCDIC value; for example, red is defined as X'F2' in the *3270 Data Stream Programmer's Reference*, and should be defined as X'32' in the EPI data stream.

- If the attribute type is greater than X'C0' (for example, field outlining), the attribute value is a binary value. The EPI uses the values defined in the *3270 Data Stream Programmer's Reference*.

Further details of 3270 orders and other control characters are supplied in the files named in the following table.

	Supplied file
COBOL copybook	cics3270.cbl
C header file	cics3270.h

Making External Security Interface calls from C and COBOL programs

You can make ESI calls from a C or COBOL Client application to verify or change passwords for a user ID known to an external security manager on a CICS server. Table 18 shows C and COBOL names that correspond to the ESI terms described in "Input and output information for ESI functions" on page 25.

Table 18. C and COBOL names corresponding to ESI terms

ESI terms	**C structure.field**	**COBOL structure.field**
Expiry date	CICS_EsiDetails_t.ExpiryDate	CICS-ESIDETAILS.EXPIRYDATE
Expiry time	CICS_EsiDetails_t.ExpiryTime	CICS-ESIDETAILS.EXPIRYTIME
Invalid count	CICS_EsiDetails_t.InvalidCount	CICS-ESIDETAILS.INVALIDCOUNT
Last access date	CICS_EsiDetails_t.LastAccessDate	CICS-ESIDETAILS.LASTACCESSDATE
Last access time	CICS_EsiDetails_t.LastAccessTime	CICS-ESIDETAILS.LASTACCESSTIME.
Last verify date	CICS_EsiDetails_t.LastVerifiedDate	CICS-ESIDETAILS.LASTVERIFIEDDATE

Table 18. C and COBOL names corresponding to ESI terms (continued)

ESI terms	C structure.field	COBOL structure.field
Last verify time	CICS_EsiDetails_t.LastVerified.Time	CICS-ESIDETAILS.LASTVERIFIEDTIME
New password	CICS_ChangePassword(,,NewPassword,,,)	CICSCHANGEPASSWORD (,,NEWPASSWORD,,,)
Old password	CICS_ChangePassword(,OldPassword,,,,)	CICSCHANGEPASSWORD (,OLDPASSWORD,,,,)
Password	CICS_VerifyPassword(,Password,,,,)	CICSVERIFYPASSWORD(,PASSWORD,,,,)
System	CICS_ChangePassword(,,,System,,)	CICSCHANGEPASSWORD(,,,SYSTEM,,)
User ID	CICS_ChangePassword(Userid,,,,,)	CICSCHANGEPASSWORD(USERID,,,,,)

Verifying a password using ESI

Use the CICS_VerifyPassword function, passing the user ID, password, and system name as input parameters. If the call is successful, the information is returned in the CICS_EsiDetails_t structure.

Changing a password using ESI

Use the CICS_ChangePassword function, passing the user ID, current password, new password, and system name as input parameters. If the call is successful, any information is returned in the CICS_EsiDetails_t structure.

Setting default security using ESI

Use the CICS_SetDefaultSecurity function, passing the user ID, password, and system name as input parameters to set the default security on a connection to a CICS server.

Compiling and linking C and COBOL applications

This section gives some examples showing how to compile and link typical ECI, EPI, and ESI applications in the various client environments. These are examples only, and may refer to specific compilers and linkers.

Refer to the samples supplied with your environment (see Appendix B, “Sample programs,” on page 197) for more information about compiling and linking programs.

For details of supported compilers, see the *CICS Transaction Gateway: Administration* book for your operating system.

Table 19 shows the header files for C required for your programs:

Table 19. C header files

Use	File
ECI	cics_eci.h
EPI	cics_epi.h
ESI	cics_esi.h
Type definitions	cicstype.h

Table 20 shows the copybook files for COBOL required for your programs:

Table 20. COBOL copybooks

Use	File
ECI	cicseci.cbl
EPI	cicsepi.cbl
ESI	cicsesi.cbl

The files contain the entry points, type definitions, data structures, and constants needed for writing programs using the ECI, EPI, and ESI interfaces.

When compiling C programs, you might need to pass structures to the external CICS interfaces in packed format. If this is the case, the C header files will contain the `#pragma pack` directive, which should not be changed.

For Micro Focus COBOL, you must use call-convention 8 for every program call, or use the default call-convention 0 and compile using the LITLINK compiler directive.

Windows

For C Programs:

- The compiler options /DWIN32, /D_WIN32, and /D_X86_=1 are used to select the correct Windows function and are standard Win32 options. These options are not specific to the CICS Transaction Gateway.
- The compiler option /DCICS_W32 must be used to define the symbol CICS_W32 to the compiler to ensure that the CICS header files are processed correctly.
- The application must be linked with the cclwin32.lib library in addition to the standard C runtime and Windows libraries.
- Callback functions must be declared using the CICSEXIT calling convention—see samples for details.

For COBOL Programs:

- It is important to use the correct calling convention when invoking the ECI or EPI from COBOL. The sample programs use the ″SPECIAL-NAMES. CALL CONVENTION 8 IS CICS.″ statements to achieve this.
- The application must be linked with the CCLWIN32.LIB library, in addition to the standard COBOL libraries, because a 32-bit Windows application is being generated.
- ECI or EPI callback functions are not supported in COBOL applications.

AIX

- The constant CICS_AIX must be defined to the compiler using the -DCICS_AIX option.
- The application must be linked with the standard AIX libpthreads.a and libc_r.a libraries, as well as the libcclaix.a library.

For COBOL Programs:

- It is important to use the correct calling convention when invoking the ECI or EPI from COBOL. When using MicroFocus COBOL the sample programs use the ″SPECIAL-NAMES CALL CONVENTION 8 IS CICS.″ statements to achieve the correct calling convention.
- To build an application, object files must be linked with the libcclaix.a library file. Only 32-bit applications are supported by the API.
- ECI or EPI callback functions are not supported in COBOL applications.

Solaris

- The constant CICS_SOL must be defined to the compiler using the -DCICS_SOL option.
- The application must be linked with the standard Solaris libpthread.so and libc.so libraries, as well as the libcclsol.so library.

Linux

General

- The constant CICS_LNX must be defined to the compiler using the –DCICS_LNX option.
- The application must be linked with the standard Linux libpthread.so and libc.so libraries, as well as the libccllnx.so library.

Linux on zSeries

The compiler option –m31 and the link option –melf_S390 must be used to build a 31-bit application. The CICS Transaction Gateway is built on a 31-bit system, so when compiling and linking applications on a 64-bit system, you must define them as 31-bit. You cannot mix 64-bit and 31-bit objects; at link stage you get incompatibility failure. When a 31-bit binary is built on a 64-bit system, all libraries must be

31-bit versions, the default pthread library is 64-bit. Typically, 31-bit libraries are installed in /lib or /usr/lib (as opposed to /lib64 and /usr/lib64, where the 64-bit versions reside).

Linux on POWER

The CICS Transaction Gateway is built on a 32-bit system, so when compiling and linking applications on a 64-bit system, you must define them as 32-bit. You cannot mix 64-bit and 32-bit objects; at link stage you get incompatibility failure. When a 32-bit binary is built on a 64-bit system, all libraries must be 32-bit versions. Typically, 32-bit libraries are installed in /lib or /usr/lib (as opposed to /lib64 and /usr/lib64, where the 64-bit versions reside).

HP-UX

- The constant CICS_HPUX must be defined to the compiler using the –DCICS_HPUX option.
- The application must be linked with the standard HPUX libpthread.sl and libc.sl libraries as well as the libcclhpux.so library.

Chapter 9. Programming in C++

This information contains information about the external access interfaces specific to C++.

Overview of the programming interface for C++

| The C++ API is not supported on Windows Vista.

Writing C++ Client applications

Establishing the working environment

You are provided with C++ (OO) support on AIX, HP-UX, Linux, Solaris and Windows operating systems. This includes the class library, C++ header files, the BMS map utility, and sample code. Note that the BMS map utility is not supported for Linux.

For full details of supported CICS servers, follow the **Support** link at the appropriate Web page:

- www.ibm.com/software/cics/ctg
- www.ibm.com/software/cics/cuc

Windows environment variables: CICS Transaction Gateway and CICS Universal Client append directories to the LIB and INCLUDE variables only in the system set, to ensure that these are used you should add `;%LIB%` to the end of your user `LIB` environment variable and `;%INCLUDE%` to the end of your user `INCLUDE` environment variable. The CICS Transaction Gateway appends `;%CLASSPATH%` to your user `CLASSPATH` environment variable. When you select **OK** on the **Edit user variable** window to make this change, the variable is displayed with the system paths appended.

Multi-threading

The CICS Transaction Gateway C++ libraries are not completely thread-safe. That is, they do not have critical sections, or semaphores, to prevent two threads from updating the same instance of an object. However, the classes do not share data, so they can be used in a well designed, multi-threaded, Client application. The normal technique is for each thread to have its own instance of lightweight objects, such as CclConn, CclFlow, CclBuf.

Making External Call Interface calls from a C++ Client program

The ECI is one of two interfaces through which a Client application can interact with a CICS server. The ECI object model consists of a set of classes which give access to the features of the ECI and supports an object-oriented approach to CICS Transaction Gateway programming with the ECI.

Linking to a CICS server program

A Client application requires one connection object, **CclConn**, for each CICS server with which it will interact. When a connection object is created, optional data can be specified which includes:

- The name of the server to be connected. This must be one of the server names defined in the configuration file *ctg.ini*. If this name is omitted, the default server will be used.
- A user ID. Some servers might require that a client application provides a user ID and password before they permit specific interactions.
- A password.

In this example, a connection object is created with a server name, user ID and password:

```
CclConn serv2( "Server2","sysad","sysad" );
```

Creating a connection object does not, in itself, cause any interaction with the server. The information in the connection object is used when one of the following server request calls is issued:

- **link—**to request the execution of a server program.
- **status—**to request the status (availability) of the server.
- **changed—**to request the notification of any change in this status.
- **cancel—**to request the cancellation of a **changed** request.

These are methods of the connection class. There are two other server request calls; the **backout** and **commit** methods of the unit of work class. More information on the use of all these methods can be found in following sections.

Passing data to a server program

A buffer object**—CclBuf** is used in the Client application to encapsulate the communication area that is used for passing data to and from a server program. The use of buffer objects is not limited to communication areas; they offer considerable flexibility for general-purpose data marshaling.

The following code constructs a buffer object and dynamically extends it as text strings are assigned, inserted and appended to its data area:

```
CclBuf comma1;
comma1 = "Some text";
comma1.insert( 9,"inserted ",5 ) += " at the end";
cout << (char*)comma1.dataArea() << endl;
  ...
```

Output produced:

```
Some inserted text at the end
```

In the next example, an existing memory structure is used. This could, for example, correspond to a record used in the server program. In this case, the buffer object knows the record is fixed-length, externally-defined, and ensures it can not be extended in any subsequent processing. The link call requests execution of the program QVALUE on the CICS server defined by the serv2 connection object and passes data via the structure on which the buffer object comma2 is overlaid.

```
struct rec{
         short key;
         char name[8];
         char retval[70];
         };
rec record1 = { 1234,"Hilary" };
CclBuf comma2( sizeof(rec),&record1 );
serv2.link( sflow,"QVALUE",&comma2 );
  ...
```

The communications area returned from a server is also contained in a buffer object.

Using COMMAREAs

A COMMAREA is a block of storage allocated by the program. The Client application uses the COMMAREA to send data to the server and the server uses the same storage to return data to the client. Therefore, you must create a COMMAREA that is large enough to contain all the information to be sent to the server and large enough to contain all the information that can be returned from the server.

For example, you need to send a 12 byte serial number to the server, but you may receive 20 Kb back from the server. You must create a COMMAREA of size 20 Kb. Your code would look like this:

```
 // serialNo is a Null terminated string
CclBuf Commarea;  // create extensible buffer object
Commarea.assign(strlen(serialNo),serialNo);  // Won't include the Null
Commarea.setDataLength(20480);  // stores Nulls in the unused area
```

In the example, the serial number is stored in the new `Commarea` which is then increased in size to 20480. The extra bytes are filled with nulls. This is important as it ensures that the information transmitted to the server is kept to a minimum. The CICS Transaction Gateway software strips off the excess nulls and transmits 12 bytes to the server.

Controlling server interactions

A flow object—**CclFlow**—controls each interaction between the Client application and a server and determines the synchronization of reply processing; synchronous, deferred synchronous or asynchronous. This example creates a synchronous flow object:

```
CclFlow sflow( Ccl::sync );
```

A flow object is referenced when a server request call is first issued and remains active from that time until all client processing of the corresponding reply from the server has been completed. At that point it is set inactive and becomes available for reuse or deletion. During its active lifespan, a flow object maintains the state of the client/server interaction it is controlling.

The flow class should be subclassed to provide the implementation of a reply handler which will be called when a reply is received; this happens regardless of the synchronization type. The reply handler is passed a buffer object which contains the communication area returned by the server. A default reply handler is provided; it just returns to the caller without doing anything.

Separate flow subclasses could be needed to cater for different client/server communication area protocols. Many flows may be active at the same time. Many servers may be used simultaneously by the same CICS Transaction Gateway or CICS Universal Client.

Managing logical units of work

A Client application uses a unit of work object, **CclUOW**, for each logical unit of work that it needs to manage. This code creates a unit of work object:

```
CclUOW uow;
```

Any server link request which participates in a unit of work references the corresponding unit of work object. When all the links participating in a unit of work have successfully completed, the unit of work can be committed by the **commit** method of the unit of work object or backed out by **backout**:

```
serv1.link( sflow, "ECITSQ", &( comma1="1st link in UOW" ), &uow );
serv1.link( sflow, "ECITSQ", &( comma1="2nd link in UOW" ), &uow );
  ...
uow.backout( sflow );
```

If no UOW object is used, each link call becomes a complete unit of work (equivalent to LINK SYNCONRETURN in the CICS server).

Whenever using logical units of work, you must ensure that you backout or commit active units of work, especially at program termination. You can check to see if a logical unit of work is still active by checking the **uowId** method of the **CclUOW** class for a non zero value.

Retrieving replies from synchronous requests

In the synchronous model, the client remains blocked at the server request call until a reply is eventually received from the server.

The example at Figure 8 calls a server program using parameters supplied on the command line. It does no subclassing to handle exceptions or to handle the reply from the server.

```
  ...
CclECI* pECI = CclECI::instance();
CclConn server1( argv[1],argv[2],argv[3] );
CclBuf  comma1( argv[4] );
CclFlow sflow( Ccl::sync );
server1.link( sflow,"ECIWTO",&comma1 );
```

Figure 8. Synchronous request to call a server program

The Client application gains access to the ECI object and constructs a connection object using the supplied server name, password and user ID. Then a buffer object is constructed using text from the command line and a synchronous flow object is created.

The link call requests execution of the CICS ECIWTO sample program on the server and passes text to it in the buffer. Processing is then blocked until a reply is received from the server. ECIWTO just writes the communication area to the operator console on the server and returns it, unchanged, to the client.

After the reply is received, the Client application reports the most recent exception code and prints the returned communication area:

```
cout << "Link returned with \""
           << pECI-> exCodeText() << "\"" << endl;
cout << "Reply from CICS server: "
           << (char*)comma1.dataArea() << endl;
```

If you call the program in Figure 8 like this:

```
ECICP01 DEVTSERV sysad sysad "Hello World"
```

the following output is expected on successful completion:

```
Link returned with "no error"
Reply from CICS server: Hello World
```

If the flow object controlling the interaction is an instance of a subclass which has implemented a reply handler, this is called and executed before processing continues with the statement following the original server request call. For example, the flow subclass defined in the asynchronous example which follows could have been used.

Retrieving replies from asynchronous requests

In the asynchronous model, the Client application issues a server request call and then continues immediately with the next statement without waiting for a reply. As soon as the reply is received from the server it is immediately passed to the reply handler of the flow object controlling the interaction; in parallel with whatever else the client happens to be doing.

The example in Figure 9 on page 131 calls a server program using parameters supplied on the command line. It subclasses the ECI class to handle exceptions and subclasses the flow class to handle the reply from the server.

Here is a simple subclass of the flow class with a reply handler implementation which just prints the reply received:

```
class MyCclFlow : public CclFlow {
public:
       MyCclFlow( Ccl::Sync sync ) : CclFlow( sync ) {}
  void handleReply( CclBuf* pcomm ){
    cout << "Reply from CICS server: "
               << (char*)pcomm-> dataArea() << endl;
    }
  };
```

A subclassed ECI object is constructed; then a connection object using the supplied server name, password and user ID. A buffer object is constructed using text from the command line and an asynchronous subclassed flow object.

The link call requests execution of the ECIWTO sample program on the server and passes text to it in the buffer object. Processing then continues with the statement following the link call:

```
  ...
MyCclECI myeci;
CclConn server1( argv[1],argv[2],argv[3] );
CclBuf  comma1( argv[4] );
MyCclFlow asflow( Ccl::async );
server1.link( asflow,"ECIWTO",&comma1 );
  ...
```

Figure 9. Asynchronous request to call a server program

In the example, there is nothing else for the main Client application to do, so to avoid premature termination, it is made to wait for user input:

```
cout << "Server call in progress. Enter q to quit..." << endl;
char input;
cin >> input;
```

Meanwhile, when the reply does come back from the server, the reply handler is called and, assuming there are no exceptions, prints the returned communication area. Note that in the asynchronous model, the buffer object to hold the returned communication area is allocated internally within the flow object, and is deleted after the reply handler has run. The buffer object supplied on the original link call is not used for the reply, and can be deleted as soon as the link call returns.

If you call the program in Figure 9 like this:

```
ecicpo2 DEVTSERV sysad sysad "Hello World"
```

the following output is expected on successful completion:

```
Server call in progress. Enter q to quit...
Reply from CICS server: Hello World
q
```

If the Client application decides at some point that it really can do no more until a reply is received from the server, it can use the **wait** method on the appropriate flow object. This effectively makes the interaction synchronous, blocking the client:

```
asflow.wait();
```

Reply solicitation calls

Deferred synchronous reply handling

In the deferred synchronous model, the Client application issues a server request call and then continues immediately with the next statement without waiting for a reply. Unlike the asynchronous case, where a server reply is handled immediately it arrives, the client decides when it wants to **poll** for a reply.

When a poll is issued, the flow object checks whether there is, in fact, a reply from the original server request. If there is, the flow object's reply handler is called synchronously and is passed the returned communication area in a buffer object. Poll returns a value to its caller indicating whether the reply was received or not; if not it can try again later.

The same simple subclass of the flow class described above is used. There are some small changes to the main Client application to indicate deferred synchronous reply handling:

```
  ...
MyCclECI myeci;
CclConn server1( argv[1],argv[2],argv[3] );
CclBuf  comma1( argv[4] );
MyCclFlow dsflow( Ccl::dsync );
server1.link( dsflow,"ECIWTO",&comma1 );
  ...
```

For demonstration purposes, the Client application is now made to loop with a delay until poll indicates the reply has been received from the server. Note that in the deferred synchronous model, a buffer object to hold the returned communication area can be supplied as a parameter to the **poll** method. If, as in the example below, no buffer object is supplied on the **poll** method, one is allocated internally within the flow object, and is deleted after the reply handler has run.

```
  ...
Ccl::Bool reply = Ccl::no;
while ( reply == Ccl::no ) {
  cout << "DSync polling..." << endl;
  reply = dsflow.poll();
  if ( reply == Ccl::no ) DosSleep( msecs );
  }
  ...
```

Typical output on successful completion would look like this:

```
DSync polling...
DSync polling...
DSync polling...
Reply from CICS server: Hello World
```

As in the asynchronous model, the **wait** method can be used to make a deferred synchronous flow synchronous, blocking the client.

ECI security

You can perform security management on servers that support Password Expiry Management (PEM). See *Supported software* in the *CICS Transaction Gateway: Administration* book for your operating system, for more information on supported servers and protocols.

To use these features you first must have constructed a Connection object. The two methods available are **verifyPassword** which checks the userid and password within the connection object with the Server Security System, and **changePassword** which allows you to change the password at the server. If successful the connection object password is updated accordingly.

If either call is successful, you are returned a pointer to an internal object which provides information about the security, a CclSecAttr object. This object provides access to information such as last verified Date and Time, Expiry Date and Time and Last access Date and Time. If you query for example last verified Date, you get back a pointer to an object which allows you to get the information in various formats. The following is a sample of code to show the use of these various objects:

```
// Connection object already created called conn
CclSecAttr *pAttrblock;            // pointer to security attributes
CclSecTime *pDTinfo;               // pointer to Date/Time information
try {
     pAttrblock = conn->verifyPassword();
     pDTinfo = pAttrblock->lastVerifiedTime();
     cout << "last verified year  :" <<pDTinfo->year() << endl;
     cout << "last verified month :" <<pDTinfo->month() << endl;
     cout << "last verified day   :" <<pDTinfo->day() << endl;
     cout << "last verified hours :" <<pDTinfo->hours() << endl;
     cout << "last verified mins  :" <<pDTinfo->minutes() << endl;
     cout << "last verified secs  :" <<pDTinfo->seconds() << endl;
     cout << "last verified 100ths:" <<pDTinfo->hundredths() << endl;
// Use a tm structure to produce a single line text of information
     tm mytime;
     mytime = pDTinfo->get_tm();
     cout << "full info:" << asctime(&mytime) << endl;
}
catch (CclException &ex)
{

// Could check for expired password error and handle if required
  cout << "Exception occurred: " <<ex.diagnose()<< endl;
}
```

Note that the security attributes and date/time memory are all handled by the connection object. If you destroy the connection object, you destroy the security information being held by that object.

Finding potential servers

Information about the CICS servers that can be used by a Client application is defined in the CICS Transaction Gateway configuration file. See *Configuration* in the *CICS Transaction Gateway: Administration* book for your operating system, for more information. The existence of such a definition doesn't guarantee availability of a server.

The ECI object **CclECI** provides access to this server information through its **serverCount**, **serverDesc**, and **serverName** methods.

Unless the ECI class has been subclassed, its unique instance is found using the class method **instance** as in the following example:

```
CclECI* pECI = CclECI::instance();
printf( "Server Count = %d\n", pECI-> serverCount() );
printf( "Server1 Name = %s\n", pECI-> serverName( 1 ) );
  ...
```

Typical output produced:

```
Server Count = 2
Server1 Name = DEVTSERV
```

Monitoring server availability

The connection object **CclConn** has three methods which can be used to determine the availability of the server connection that it represents:

status requests the status (that is, the availability) of the server.

changed
requests notification of any change in this status.

cancel
requests cancellation of a **changed** request.

The example described below shows how server availability can be monitored in a Client application that is busy doing something else.

Here is a subclass of the flow class designed for use with server status calls. The reply handler implementation prints the server name and its newly-changed status; it ignores the returned communication area. Next, it issues a changed server request so that the next server status change will be received. The reply handler will be called every time the availability of the server changes.

```
class ChgFlow : public CclFlow {
public:
       ChgFlow( Ccl::Sync stype ) : CclFlow( stype ) {}
  void handleReply( CclBuf* ) {
         CclConn* ccon = connection();
         cout << ccon-> serverName() << " is "
              << ccon-> serverStatusText() << endl;
         ChgFlow* sflow = new ChgFlow( Ccl::async );
         ccon-> changed( *sflow );
         }
  };
```

The main Client application iterates through all the servers listed in the CICS Transaction Gateway Initialization file. For each one, an asynchronous status request call is issued. The Client application continues with whatever else it has to do.

```
int numservs = myeci.serverCount();
CclConn* pcon;
ChgFlow* pflo;
for ( int i = 1; i <= numservs ; i++ ) {
  pcon = new CclConn( myeci.serverName( i ) );
  pflo = new ChgFlow( Ccl::async );
  pcon-> status( *pflo );
  }
  ...
```

The output produced could look something like this:

```
PROD1    is unavailable
DEVTSERV is unavailable
PROD1    is available
```

Initially, both servers are unavailable because the ECI Client application is not running. It starts, and after a while makes contact with one of the servers.

C++ ECI classes

Table 21summarizes the classes provided for programming using the C++ interface:

Table 21. C++ ECI classes.

Object	Classname	Description
Global	Ccl	Contains global enumerations
Buffer	CclBuf	Used for exchanging data with a server
Connection	CclConn	Models the connection to a server
ECI	CclECI	Controls and lists access to CICS servers
Exception	CclException	Encapsulates exception information
Flow	CclFlow	Handles a single client/server interaction
SecAttr	CclSecAttr	Provides information about security attributes (passwords)
SecTime	CclSecTime	Provides date and time information
UOW	CclUOW	Corresponds with a Unit of Work in the server—used for managing updates to recoverable resources.

Making External Presentation Interface Calls from a C++ Client Program

In procedural programming, the External Presentation Interface (EPI) provides a mechanism for clients to communicate with transactions on a server and to handle 3270 data streams.

The classes provided to support the EPI make it simpler for a programmer using OO techniques to access the facilities that EPI provides:

- Connection of 3270 sessions to CICS servers
- Starting CICS transactions
- Sending and receiving 3270 data streams

The classes also enhance the procedural CICS EPI support by providing higher level constructs for handling 3270 data streams:

1. General purpose C++ classes for handling 3270 data stream, such as fields and attributes, and CICS transaction routing data, such as transaction ID.
2. Generation of C++ classes for specific CICS applications from BMS map source files. These classes allow client applications to access data on 3270 panels, using the same field names as used in the CICS server BMS application.

Note: These classes do not contain any specific support for 3270 data streams that contain DBCS fields. Data streams with a mixture of DBCS and SBCS fields are not supported.

The BMS utility is a tool for statically producing C++ class source code definitions and implementations from a CICS BMS mapset.

Note: CICSBMSC is not provided with CICS Transaction Gateway for the Linux operating system.

Adding a terminal to CICS

The EPI must be initialized, by creating a CclEPI object, before a terminal connection can be made to CICS. The CclEPI object, like the CclECI object, also provides access to information about CICS servers which have been configured in the CICS Transaction Gateway configuration file. The following C++ sample shows the use of the CclEPI object:

```
#include <cicsepi.hpp> // CICS Transaction Gateway EPI headers
  ...
CclEPI epi; // Initialize CICS Transaction Gateway EPI
// List all CICS servers in Gateway initialization file
for ( int i=1; i<= EPI.serverCount(); i++ )
    cout << EPI.serverName(i) << "  "
         << EPI.serverDesc(i) << endl;
```

To add a 3270 terminal to CICS, a CclTerminal object is created. The CICS server name used must be configured in the CICS Transaction Gateway initialization file. To start a transaction on the CICS server a CclSession object is required to control the session. The required transaction (in this example the CICS-supplied sign-on transaction CESN) can then be started using the **send** method on the CclTerminal object:

```
try {
    // Connect to CICS server
    CclTerminal terminal( "CICS1234" );
    // Start CESN transaction on CICS server
    CclSession session( Ccl::sync );
    terminal.send( &session, "CESN" );
    ...
} catch ( CclException &exception ) {
    cout << "CclClass exception: " << exception.diagnose() << endl;
}
```

Note the use of try and catch blocks to handle any exceptions thrown by the CICS classes.

EPI call synchronization types

The EPI C++ classes support synchronous (“blocking”), and deferred synchronous (“polling”) and asynchronous (“callback”) protocols.

In the example above the CclSession object is created with the synchronization type of **Ccl::sync**. When this CclSession object is passed as the first parameter on a CclTerminal **send** method, a synchronous call is made to CICS. The C++ Client application is then blocked until the reply was received from CICS. When the reply is received, updates are made to the CclScreen object according to the 3270 data stream received, then control is returned to the C++ program.

To make asynchronous calls the CclSession object used on the CclTerminal **send** method is created with a synchronization type of **Ccl::async**. The call is made to CICS using the CclTerminal **send** method, but control returns immediately to the Client application without waiting for a reply from CICS. The CclTerminal object starts a separate thread which waits for the reply from CICS. When a reply is received, the **handleReply** method on the CclSession object is invoked. To process the reply, the **handleReply** method should be overridden in a **CclSession** subclass:

```
class MySession : public CclSession {
public:
    MySession(Ccl::Sync protocol) : CclSession( protocol ) {}
    // Override reply handler method
    void handleReply( State state, CclScreen* screen );
};
```

The implementation of the **handleReply** method can process the screen data available in the CclScreen object, which will have been updated in line with the 3270 data stream sent from CICS:

```
void MySession::handleReply( State state, CclScreen* screen ) {
    // Check the state of the session
    switch( state ) {
    case CclSession::client:
    case CclSession::idle:
        // Output data from the screen
        for ( int i=1; i < screen->fieldCount(); i++ ) {
            cout << "Field " << i << ": " << screen->field->text();
        screen->setAID( CclScreen::PF3 );
        ...
    } // end switch
}
```

The **handleReply** method is called for each transmission received from CICS. Depending on the design of the CICS server program, a CclTerminal **send** call may result in one or more replies. The *state* parameter on the **handleReply** method indicates whether the server has finished sending replies:

CclSession::server
indicates that the CICS server program is still running and has further data to send. The Client application can process the current screen contents immediately, or simply wait for further replies.

CclSession::client
indicates that the CICS server program is now waiting for a response. The Client application should process the screen contents and send a reply.

CclSession::idle
indicates that the CICS server program has completed. The Client application should process the screen contents and either disconnect the terminal, or start a further transaction.

Most Client application will want to wait until the CICS server program has finished sending data (that is, the **CclSession**/**CclTerminal** state is client or idle) before processing the screen. However, some long-running server programs may send intermediate results or progress information that can usefully be accessed while the state is still server.

The implementation of the **handleReply** method can read and process data from the **CclScreen** object, update fields as required, and set the cursor position and AID key in preparation for the return transmission to CICS. The Client application main program should invoke further methods (**send** or **disconnect**) on the **CclTerminal** object to drive the server application:

```
try {
    // Connect to CICS server
    CclTerminal terminal( "CICS1234" );
    // Create asynchronous session
    MySession session(Ccl::async);
    // Start CESN transaction on CICS server
    terminal.send( &session, "CESN" );
    // Replies processed asynchronously in overridden
    // handleReply method
    ...
} catch ( CclException &exception ) {
    cout << "CclClass exception: " << exception.diagnose() << endl;
}
```

Note that the **handleReply** method is run on a separate thread. If the main Client application program needs to know when the reply has been received, a message or semaphore could be used to communicate between the **handleReply** method and the main program.

To make deferred synchronous calls the CclSession object used on the **CclTerminal send** method is created with a synchronization type of **Ccl::dsync**. As in the asynchronous case, a call is made to CICS using the **CclTerminal send** method and control returns immediately to the Client application without waiting for a reply from CICS. 3270 screen updates from CICS must be retrieved later using the **poll()** method on the Terminal object:

```
try {
    // Connect to CICS server
    CclTerminal terminal( "CICS1234" );
    // Create deferred synchronous session
    MySession session(Ccl::dsync);
    // Start CESN transaction on CICS server
    terminal.send( &session, "CESN" );
    ...
    if ( terminal.poll())
        // reply processed in handleReply method
    else
        // no reply received yet
} catch ( CclException &exception ) {
    cout << "CclClass exception: " << exception.diagnose() << endl;
}
```

A CICS server transaction may send more than one reply in response to a **CclTerminal send** call. More than one **CclTerminal poll** call may therefore be needed to collect all the replies. Use the **CclTerminal state** method to find out if further replies are expected. If there are, the value returned will be server.

As in the synchronous and asynchronous cases, the **handleReply** method can conveniently be used to encapsulate the code processing the 3270 data returned from CICS from one or more transmissions.

Sending and receiving data

Accessing fields on CICS 3270 screens

Once a terminal connection to CICS has been established, the CclTerminal, CclSession, CclScreen and CclField objects are used to navigate through the screens presented by the CICS server application, reading and updating screen data as required.

The CclScreen object is created by the CclTerminal object and is obtained via the **screen** method on the CclTerminal object. It provides methods for obtaining general information about the 3270 screen (e.g. cursor position) and for accessing individual fields (by row/column screen position or by index). The following example prints out field contents, then ends the CESN transaction (started above) by returning **PF3**:

```
// Get access to the CclScreen object
CclScreen* screen = terminal.screen();
for ( int i=1; i ≤ screen->fieldCount(); i++ ) {
    CclField* field = screen->field(i); // get field by index
    if ( field->textLength > 0 )
        cout << "Field " << i << ": " << field->text();
}
// Return PF3 to CICS
screen->setAID( CclScreen::PF3 );
terminal.send( &session );
// Disconnect the terminal from CICS
terminal.disconnect();
```

The **CclField** class provides access to the text and attributes of an individual 3270 field. These can be used in a variety of ways to locate and manipulate information on a 3270 screen:

```
for ( int i=1; i ≤ screen->fieldCount(); i++ ) {
    CclField* field = screen->field(i); // get field by index
    // Find unprotected (i.e. input) fields
    if ( field->inputProt() == CclField::unprotect )
        ...
    // Find fields containing a specific text string
    if ( strstr(field->text(), "CICS Sign-on") )
        ...
    // Find red fields
    if ( field->foregroundColor() == CclField::red )
        ...
}
```

Note that the string “Sign-on” in the above sample may need to be changed to meet local conventions. For example, an AIX server may use the string “SIGNON”.

Converting BMS maps and using the Map class

A large proportion of existing CICS applications use BMS maps for 3270 screen output. This means that the server application can use data structures corresponding to named fields in the BMS map rather than handling 3270 data stream directly. The EPI BMS conversion utility uses the information in the BMS map source to generate classes specific to individual maps, that allow fields to be accessed by their names, and allow field lengths and attributes to be known at compile time.

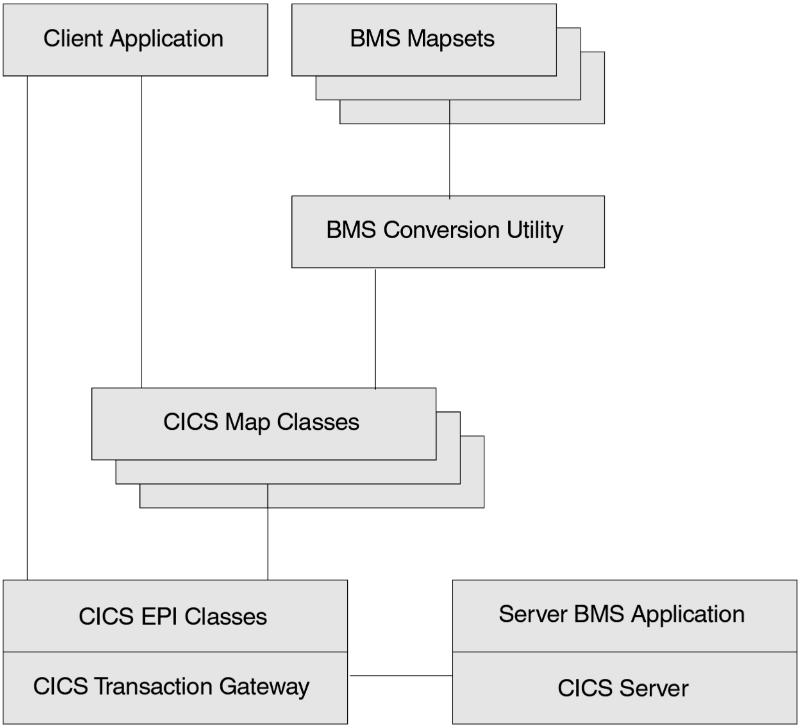


Figure 10. Use of BMS map classes

The utility generates C++ class definitions and implementations that applications can use to access the map data as named fields within a map object. A class is defined for each map, allowing field names and lengths to be known at compile time. The C++ classes use the CICS EPI base classes to handle the inbound and outbound 3270 data streams. The generated classes inherit a base class **CclMap** that provides general functions required by all map classes.

Run the CICSBMSC utility on the BMS source as follows:

```
CICSBMSC <filename>.BMS
```

See the note at "Making External Presentation Interface Calls from a C++ Client Program" on page 136 for BMS support on Linux.

The utility generates `.HPP` and `.CPP` files containing the definition and implementation of the map classes.

Having used the EPI BMS utility to generate the map class, use the base EPI classes to reach the required 3270 screens in the usual way. Then use the map classes to access fields by their names in the BMS map. The map classes are validated against the data in the current **CclScreen** object.

Mapset containing a single map

The mapset listed in Figure 11 contains a simple map, MAPINQ1.

```
***************************************************************
*  cicssda  MAPINQ1  -- Wed  2 Aug 14:14:02 1995
***************************************************************
MAPINQ1 DFHMSD TYPE=&SYSPARM,MODE=INOUT,LANG=C,STORAGE=AUTO,TIOAPFX=YES
MAPINQ1 DFHMDI SIZE=(24,80),MAPATTS=(COLOR,HILIGHT,VALIDN),LINE=1,      X
               COLUMN=1,COLOR=NEUTRAL,HILIGHT=OFF
DTITLE  DFHMDF POS=(2,2),LENGTH=5,ATTRB=(PROT,NORM),COLOR=TURQUOISE,   X
               CASE=MIXED,INITIAL='Date:'
DATE    DFHMDF POS=(2,9),LENGTH=8,ATTRB=(PROT,BRT),CASE=MIXED
...
PRODNAM DFHMDF POS=(5,24),LENGTH=40,ATTRB=(PROT,BRT),CASE=MIXED
...
APPLID  DFHMDF POS=(15,15),LENGTH=8,ATTRB=(PROT,BRT),CASE=MIXED
...
MAPINQ1 DFHMSD TYPE=FINAL
```

Figure 11. Sample Map Class—BMS Source

The BMS Conversion Utility generates the C++ class definition (shown in Figure 12 on page 144) from this mapset. The class name “MAPINQ1Map” is derived from the map name in the BMS source. The class inherits the **CclMap** class.

The class provides these main operations:

1. The constructor **MAPINQ1Map** invokes the parent constructor, that validates the map object against the current screen.
2. The method **field** provides access to fields in the map, using the BMS source field names (provided as an enumeration within the class).

```
//************* CICS Transaction Gateway Classes ************************************
//
// FILE NAME:   epiinq.hpp
//
// DESCRIPTION: C++ header for epiinq.bms
//              Generated by CICS BMS Conversion Utility - Version 1.0
//
//**********************************************************************
#include <cicsepi.hpp>               // CICS Transaction Gateway EPI classes
//------------------------------------------------------------------------
// MAPINQ1Map class declaration
//------------------------------------------------------------------------
class MAPINQ1Map : public CclMap {
public:
  enum           FieldName {
                   DTITLE,
                   DATE,
                   ...
                   PRODNAM,
                   ...
                   APPLID,
                   ...
                 };
//-------------- Constructors/Destructors -------------------------------
                 MAPINQ1Map( CclScreen* screen );
                 ~MAPINQ1Map();
//-------------- Actions ------------------------------------------------
  CclField*      field( FieldName name );        // access field by name
...
}; // end class
```

Figure 12. Sample Map Class—Generated C++ Header

Using EPI BMS Map Classes

The map classes generated using `CICSBMSC` can be compiled and built into a Client application. Note that when building Windows applications using pre-compiled headers, add `#include stdafx.h` to the `.cpp` file generated by `CICSBMSC`.

CclEPI, **CclTerminal** and **CclSession** objects are used in the normal way to start a CICS transaction:

```
try {
   // Initialize CICS Transaction Gateway EPI
   CclEPI epi;
   // Connect to CICS server
   CclTerminal terminal( "CICS1234" );
   // Start transaction on CICS server
   CclSession session( Ccl::sync );
   terminal.send( &session, "EPIC" );
```

In this example the server program uses a BMS map for its first panel, for which a map class “MAPINQ1Map” has been generated. When the map object is created, the constructor validates the screen contents with the fields defined in the map. If validation is successful, fields can then be accessed using their BMS field names instead of by index or position from the **CclScreen** object:

```
    MAPINQ1Map map( terminal.screen() );
    CclField* field;
    // Output text from "PRODNAM" field
    field = map.field(MAPINQ1Map::PRODNAM);
    cout << "Product Name: " << field->text() << endl;
    // Output text from "APPLID" field
    field = map.field(MAPINQ1Map::APPLID);
    cout << "Product Name: " << field->text() << endl;
} catch (CclException &exception) {
    cout << exception.diagnose()<<endl;
}
```

BMS Map objects can also be used within the **handleReply** method for asynchronous and deferred synchronous calls.

For validation to succeed, the entire BMS map must be available on the current screen. A map class cannot therefore be used when some or all of the BMS map has been overlayed by another map or by individual 3270 fields.

Support for Automatic Transaction Initiation (ATI)

Client applications can control whether ATI transactions are allowed by using the setATI() and queryATI() methods on the **CclTerminal** class. The default setting is for ATIs to be disabled. The following code fragment shows how to enabled ATIs for a particular terminal:

```
// Create terminal connection to CICS server
CclTerminal terminal( "myserver" );
// Enable ATIs
terminal.setATI(CclTerminal::enabled);
```

The CclTerminal class performs one or more of the following

- Run any outstanding ATIs as soon as a transaction ends
- Call additional programming needed to handle the ATI replies
- Run ATIs before or between client-initiated transactions

depending on whether the call synchronization type is Synchronous, Asynchronous or Deferred synchronous.

Synchronous

When you call the **CclTerminal send()** method, any outstanding ATIs will be run after the client-initiated transaction has completed. The CclTerminal class will wait for the ATI replies then update the **CclScreen** contents as part of the synchronous send() call. If you expect an ATI to occur before or between client-initiated transactions, you can call the **CclTerminal** receiveATI() method to wait synchronously for the ATI.

Asynchronous

When the client application calls the **CclTerminal** send() method for an async session, the **CclTerminal** class starts a separate thread to handle replies. If ATIs are disabled, this thread finishes when the CICS transaction is complete. If ATIs are enabled, the reply thread continues to run between transactions. When the **CclTerminal** state becomes idle, any outstanding ATIs are run and ATIs received subsequently are run immediately. The reply thread is not started until the first `CclTerminal::send()` call, so if you expect ATIs to occur before any client-initiated transactions, you can call the **receiveATI()** method to start the reply thread.

Deferred synchronous

After the **CclTerminal send()** method is called for a dsync session, the poll() method is used to receive the replies. Outstanding ATIs are started when the last reply has been received (i.e. on the final poll() call). You can also call the poll() method to start and receive replies for ATIs between client-initiated transactions. As the poll() method can be called before or between client-initiated transactions, the receiveATI() method is not needed (and is invalid) for deferred synchronous sessions. For any of the synchronization types you can provide a handleReply() method by subclassing the CclSession class. As for client-initiated transactions, this method will be called when the ATI 3270 data has been received and the CclScreen object updated. The transID() method on the CclTerminal or CclSession can be called to identify the ATI.

EPI Security

You can perform security management on servers that support Password Expiry Management (PEM). See *Supported software* in the *CICS Transaction Gateway: Administration* book for your operating system, for more information on supported servers and protocols.

To use these features you first must have constructed a **CclTerminal** object which is sign-on incapable, in other words it must have a userid and password (even if they are null). The two methods available are **verifyPassword** which checks the userid and password within the terminal object with the Server Security System, and **changePassword** which allows you to change the password at the server. If successful the connection object password is updated accordingly.

If either call is successful, you are returned a pointer to an internal object which provides information about the security, a **CclSecAttr** object. This object provides access to information such as last verified Date and Time, Expiry Date and Time and Last access Date and Time. If you query for example last verified Date, you get back a pointer to an object which allows you to get the information in various formats. The following is sample code to show the use of these various objects.

```
// Terminal object already created called term
CclSecAttr *pAttrblock;           // pointer to security attributes
CclSecTime *pDTinfo;              // pointer to Date/Time information
try {
     pAttrblock = term->verifyPassword();
     pDTinfo = pAttrblock->lastVerifiedTime();
     cout << "last verified year  :" <<pDTinfo->year() << endl;
     cout << "last verified month :" <<pDTinfo->month() << endl;
     cout << "last verified day   :" <<pDTinfo->day() << endl;
     cout << "last verified hours :" <<pDTinfo->hours() << endl;
     cout << "last verified mins  :" <<pDTinfo->minutes() << endl;
     cout << "last verified secs  :" <<pDTinfo->seconds() << endl;
     cout << "last verified 100ths:" <<pDTinfo->hundredths() << endl;

// Use a tm structure to produce a single line text of information

     tm mytime;
     mytime = pDTinfo->get_tm();
     cout << "full info:" << asctime(&mytime) << endl;
}
catch (CclException &ex)
{

// Could check for expired password error and handle if required
   cout << "Exception occurred: " <<ex.diagnose()<< endl;
}
```

Note that the security attributes and date/time memory are all handled by the terminal object. If you destroy the terminal object, you destroy the security information being held by that object.

C++ EPI classes

Table 22 summarizes the C++ EPI classes. For full details of the methods each class provides, refer to the C++ chapter, in *CICS Transaction Gateway: Programming Reference*.

Table 22. C++ EPI classes.

Object	Classname	Description
Global	Ccl	Contains global enumerations.
EPI	CclEPI	Initializes the EPI. This class also has methods that obtain information on CICS servers accessible to the CICS Transaction Gateway or CICS Universal Client.
Exception	CclException	Encapsulates error information.

Table 22. C++ EPI classes. (continued)

Object	**Classname**	**Description**
Field	CclField	Supports a single field on a virtual screen and provides access to field text and attributes.
Map	CclMap	This class provides access to **CclField** objects, using BMS map information. The CICSBMSC utility generates classes derived from **CclMap**. See the note at "Making External Presentation Interface Calls from a C++ Client Program" on page 136 for BMS support on Linux.
Screen	CclScreen	Each terminal (**CclTerminal** object) has a virtual screen associated with it. The **CclScreen** class contains a collection of **CclField** objects and methods to access these objects. It also has methods for general screen handling.
SecAttr	CclSecAttr	Provides information about security attributes (passwords)
SecTime	CclSecTime	Provides date and time information
Session	CclSession	Controls communication with the server in synchronous, asynchronous and deferred synchronous modes. Applications can use **CclSession** to derive their own classes to encapsulate specific CICS transactions.
Terminal	CclTerminal	Controls a 3270 terminal connection to CICS. The **CclTerminal** class handles CICS conversational, pseudo-conversational, and ATI transactions. One application can create many **CclTerminal** objects.

Compiling and running a C++ Client application

Refer to the sample programs for more information about compiling and linking programs; see Appendix B, "Sample programs," on page 197.

Your C++ program source needs #include statements to include either cicseci.hpp, for the ECI classes, or cicsepi.hpp, for the EPI classes. These files are in the <install_path>\include subdirectory on Windows or the <install_path>/ include subdirectory on UNIX and Linux.

Define the following macros on UNIX and Linux operating systems, when compiling C++ applications that use the CICS C++ libraries.

Operating system	**Macro**
AIX	CICS_AIX
HP-UX	CICS_HPUX
Linux	CICS_LNX
Solaris	CICS_SOL

On HP-UX Itanium® hardware all C++ applications must be compiled with the -AP flag in order to run successfully with the CICS Transaction Gateway, for example:

```
aCC -AP -DCICS_HPUX file.cpp
```

On Windows operating systems, the CICS Transaction Gateway API DLL is built using the synchronous model of C++ exception handling which assumes that external C functions do not throw exceptions. This support is true for both Microsoft Visual C++ .NET 2003 and Microsoft Visual C++ 2005 compilers.

Problem determination for C++ Client programs

Handling Exceptions

Most class methods could generate an exception. The default exception handler is found in the handleException method in the **CclECI** and **CclEPI** classes. It is a simple routine which does a C++ throw of a **CclException** object. It does not perform any action if an exception occurs within the destruction of an object. You must not do a throw within a destructor as this causes unpredictable results.

This routine is suitable for most needs when using synchronization modes of dsync and sync. For example:

```
#include <iostream.h>
#include <cicseci.hpp>

void main(void) {
  CclECI *eci;
  eci = CclECI::instance();
  CclFlow flow(Ccl::sync);
  CclBuf buf;
  CclConn conn("CICSOS2","SYSAD","SYSAD");
  buf.setDataLength(80);
  try {
    conn.link(flow,"EC01",&buf);
    cout << (char *)buf.dataArea() << endl;
  }
  catch(CclException &exc) {
    cout << "link failed" << endl;
    cout << "diagnose:" << exc.diagnose() << endl;
    cout << "abend code:" << exc.abendCode() << endl;
  }
};
```

You might want to implement your own exception handler, by subclassing the CclECI or CclEPI class, if you want to handle object destruction exceptions explicitly.

```
void CclECI::handleException(CclException except) {
  if (*(except.methodName()) != '~') {
    throw( except );
  }   else {

// Handle a destructor exception, but ensure that this
// routine just returns

  }
};
```

Async Exception Handling

You must override the ECI handleException routine by subclassing CclECI if you are using the async synchronization mode. With async mode a separate thread controlled by the class library dll is created and an exception can occur on that thread. If an exception does occur on that thread, the default exception handler would throw the exception but there is no code in the class library to trap the throw. For unhandled exceptions, the default action of most compilers' runtimes is to terminate the application.

To create a new exception handler you do the following

```
class MyCclECI : public CclECI {
public:
   void handleException( CclException ex) {
         // Place whatever code you want here, for example set a
         // semaphore, or generate a Window Message
         }
   };
```

Once you have subclassed the ECI Class, you still can only create one object of this class for your application, however do not use the instance method, you must create the object either explicitly e.g.

```
MyCclECI myeci;
```

or by using the new operator

```
MyCclECI *pmyeci;
pmyeci = new MyCclECI;
```

Chapter 10. Programming in COM

This contains information about the external access interfaces specific to COM.

Overview of the programming interface for COM

COM classes are provided for the CICS ECI, EPI, and ESI functions on Windows. These COM classes are supported only for use with Visual Basic and VBScript.

Writing COM Client applications

Establishing the working environment

You are provided with Component Object Model (COM) Object Oriented (OO) support for Client applications in the Windows environment. This includes the COM runtimes, type libraries, the BMS map utility, and sample code.

The COM libraries: The COM libraries are automation compatible.

Servers:

The libraries are provided as in-process servers (cclieci.dll and ccliepi.dll).

Registration:

The COM libraries are registered at installation time. This includes the COM classes, associated ProgIDs and the type libraries.

Visual Basic will only use the type libraries if you register them to each Visual Basic Project. It is recommended that you do this to make full use of the features and performance enhancements of these type libraries. See “Enabling the use of the COM libraries” on page 154 for details about project enablement.

VBScript does not use type libraries.

All the COM libraries support automatic registration and de-registration.

Use the Microsoft supplied program REGSVR32 to register or de-register a server.

For example to register or reregister the ECI COM libraries issue the command:

```
REGSVR32 CCLIECI.DLL
```

to de-register issue the command

```
REGSVR32 /U CCLIECI.DLL
```

Enabling the use of the COM libraries: To set up Visual Basic to use the type libraries, go to the Visual Basic Project/References... dialog and select either EPI or ECI depending on your application needs.

If the type libraries are not listed then the COM libraries probably have not been registered. Refer to the previous section for information on registering the COM libraries.

COM Libraries: Objects and Apartments: The design of the Client COM Libraries requires the passing of a COM object to another COM object. For this to work the relevant COM objects need to be created in the same apartment. For example, in ECI, to make a link method call on the Connect COM object a Flow, Buffer and UOW object need to be passed. These must all be created in the same apartment in order to function properly.

Again with EPI it is important to ensure that the Terminal Session and Map COM Objects are created in the same apartment. The Terminal is responsible for creating the Screen object and it will create it in the same apartment as itself. This Screen object is then responsible for creating field objects and also creates them in the same apartment as itself. The programmer has control of the apartment where COM objects are created.

In most cases in Visual Basic you do not need to worry about apartments as you will be creating single threaded applications.

Object Creation and Interfaces

To talk to COM objects you have to use interfaces. The ECI and EPI COM libraries provide two interfaces per COM class.

The first interface is called IDispatch and is provided to support old Visual Basic applications and VBScript. A second interface, a Custom interface, is also provided for use by Visual Basic. This interface is faster than the IDispatch interface and it is recommended that you use this interface with Visual Basic. Each COM class provides an IDispatch interface and a Custom interface.

Visual Basic provides more than one way to create a COM object and select the interface to talk to that object. To create an object there are the CreateObject function and the New function. It is recommended that you use the New function to create objects in Visual Basic.

VBScript is simpler. It provides only one way to create an object, the CreateObject function, and you must use the IDispatch interface.

The following are some examples of creating COM objects

```
Set eci = CreateObject("Ccl.ECI")
Set eci = New CcloECI
Set connection = CreateObject("Ccl.Connect")
Set connection = New CcloConn
```

Note the two ways you can request the object class. When using CreateObject you specify a string called the Programmatic ID or ProgID for short. When using the New function you specify the Class name that is registered in the type library.

When using Visual Basic you have the choice of which interface you want to use. If you DIM your variable as Object, then you select the IDispatch interface. If you DIM your variable as the Class name then you will select the custom interface. To create a terminal object in Visual Basic you would use the code:

```
Dim Terminal as CclOTerminal
Set Terminal = New CclOTerminal
```

Figure 13. Creating a terminal object in Visual Basic

or you can combine the above into a single statement if you wish

```
Dim Terminal as New CclOTerminal
```

When using VBScript, VBScript will automatically select the IDispatch interface for you. For example to create a terminal Object in VBScript you would use the code

```
Dim Terminal
Set Terminal = CreateObject("Ccl.Terminal")
```

Figure 14. Creating a terminal Object in VBScript

It is recommended that you:

- choose one interface type or the other.
- do not mix the object interface types in your program. This type of environment is not supported.
- select the custom interface because it should provide performance improvements.

No matter which interface you select or how the object is created, you use the objects identically in your program.

Type Libraries and Visual Basic Intellisense

Type libraries add many useful features to the COM libraries. One of these is Visual Basic Intellisense. The type libraries provide Visual Basic with information so that it can help you with code completion. It prompts you with the format of the method and, where applicable, constants which might be relevant to method parameters or return values you can test for. For example if you create a terminal object for Visual Basic as shown in Figure 13 on page 155, when you want to select a method on the terminal object, press the '.' key and you are presented with a list of available methods. Select the method and press space or open bracket and you are shown the required parameters. You can also browse the type libraries for reference information on the ECI and EPI classes by using the Visual Basic Object Browser. Select either CclECILib for ECI classes reference or CclEPILib for EPI classes reference information. The type libraries are embedded within the in-process library files cclieci.dll and ccliepi.dll.

Making External Call Interface calls from a COM Client program

Linking to a CICS server program using Visual Basic

The first step is to declare object variables for the ECI interfaces to be used. See the *COM* chapter of *CICS Transaction Gateway: Programming Reference* for details of the available interfaces. Declarations are usually made in the General Declarations section of a Visual Basic program:

```
Dim ECI As CclOECI
Dim Connect As CclOConn
Dim Flow As CclOFlow
Dim Buffer As CclOBuf
Dim UOW As CclOUOW
```

The required ECI objects are then instantiated using the Visual Basic **New** function. This can be done in the **Form_Load** subroutine or at some later stage in response to some user action. Note that a CclOECI object must be created first.

```
Sub ECILink_Click()
    Set ECI = New CclOECI
    Set Connect = New CclOConn
    Set Flow = New CclOFlow
    Set Buffer = New CclOBuf
```

Details of the CICS server to be used – server name (as configured in the Gateway initialization file) , userid and password – are supplied via the **Details** method on the Connect object. The Buffer object is initialized with some data to be sent to CICS:

```
    Connect.Details "CICSNAME", "sysad", "sysad"
    Buffer.SetString "Hello"
```

Now we are ready to make the call to CICS. The **Link** method takes as parameters the Flow object, the name of the CICS server program to be invoked, the Buffer object and a UOW object. In this example a null variable is supplied for the UOW parameter, so this call will not be part of a recoverable Unit Of Work. The contents of the Buffer returned from CICS are output to a Visual Basic text box “Text1”:

```
    Connect.Link Flow, "ECIWTO", Buffer, UOW
    Text1.Text = Buffer.String
```

Finally the CICS COM objects are deleted:

```
    Set Connect = Nothing
    Set Flow = Nothing
    Set Buffer = Nothing
  End Sub
```

This example sends and receives a simple text string. In practice, the Buffer object would contain more complex data (for example C data structure). For binary data the **Buffer.SetData** and **Buffer.Data** methods are provided to allow the contents to be accessed as a Byte array.

A typical client application could access CICS through one or more **Connect.Link** calls and construct a ’business object’ for use in end-user Basic programs. One approach to this would be to implement the ’business object’ as a separate COM automation server containing the logic to process the contents of the CclOBuffer objects.

Handling COMMAREAs in Visual Basic

A CommArea is a block of storage that contains all the information you send to and receive from the server. Because of this, you must create a CommArea that is big enough for this information. For example, you might need to send a 12 byte serial number to the server, but receive a maximum of 20 Kb back from the server; this means you must create a Commarea of size 20 Kb. To do this you could code

```
Set Buf = new CclOBuf  ' create extensible buffer object
Buf.SetString(serialNo)
Buf.setLength(20480)   ' stores Nulls in the unused area
```

In the above example, Commarea is given the serial number and the buffer is increased to the required amount, but the extra area is filled with nulls. This is important as it ensures that the information transmitted to the server is kept to a minimum. The Client daemon strips off the excess nulls and only transmits the 12 bytes to the server.

Linking to a CICS server program using VBScript

This is similar to the previous section visual basic but the creating of the objects is different.

It is not necessary to DIM any variables with VBScript but it would be good programming practice to do so.

```
Dim ECI, Connect, Flow, Buffer, UOW
```

To create the objects you use the code:

```
Set ECI = CreateObject("Ccl.ECI")
Set Connect = CreateObject("Ccl.Connect")
Set Flow = CreateObject("Ccl.Flow")
Set Buffer = CreateObject("Ccl.Buffer")
Set UOW = Nothing
```

If you are not going to use a UOW, you must explicitly set it to ’Nothing’ in VBScript.

Managing a LUW

ECI Link Calls within a Unit Of Work

Using the **UOW** COM class, a number of link calls can be made to a CICS server within a single Unit of Work. Updates to recoverable resources in the CICS server can then be committed or backed out by the client program as necessary.

In this example a UOW object is created, and is used as a parameter to the **Connect.Link** calls:

```
Sub ECIStartUOW_Click()
    'Instantiate CICS ECI objects
    Set Connect = New Ccl0Conn
    Set Flow = New Ccl0Flow
    Set UOW = New Ccl0UOW
    Set Buffer = New Ccl0Buf
    Connect.Details "CICSNAME", "sysad", "sysad"
End Sub

Sub ECILink_Click()
    'Set up the commarea buffer
    Buffer.SetString Text1.Text
    Buffer.SetLength 80
    'Make the link call as part of a Unit of Work
    Connect.link Flow, "ECITSQ", Buffer, UOW
End Sub
```

After a number of link calls have been made, the **Commit** or **Backout** methods on the **Ccl UOW** interface can be used:

```
Sub Commit_Click()
    'Commit the CICS updates
    UOW.Commit Flow
End Sub
Sub Backout_Click()
    'Backout the CICS updates
    UOW.Backout Flow
End Sub
```

If no UOW object is used (a NULL value is supplied on the **Connect.Link** call), each link call becomes a complete unit of work (equivalent to LINK SYNCONRETURN in the CICS server).

When you use Logical units of work, you must ensure that you backout or commit active units of work, this is particularly important at program termination. You can check if a logical unit of work is still active by checking the uowId method for a non-zero value.

In Visual Basic, if you Dim a UOW variable but never create the object, it is assumed to a be of value **Nothing** and the Link call will therefore not associate a unit of work with the call. In VBScript, however, it is necessary to ensure explicitly that the variable is set to nothing. To do this code

```
Set UOW=Nothing
```

before making your link call.

Retrieving replies from asynchronous requests

The Client daemon ECI COM classes support synchronous ("blocking") and deferred synchronous ("polling") protocols. These classes do not support the asynchronous calls that are available in the C++ classes.

Reply solicitation calls

Deferred synchronous reply handling: In the examples in section "Linking to a CICS server program using Visual Basic" on page 156 a Flow object was used with the default synchronization type of cclSync. When this Flow object was used as the first parameter on **Connect.Link**, a synchronous link call was made to CICS. The Visual Basic program was then blocked until the reply was received from CICS. When the link call returned the reply from CICS was immediately available in the Buffer object.

To make a deferred synchronous call you use the **SetSyncType** method on the Flow object to set the Flow to cclDSync. When this Flow object is used on a **Connect.Link** call, the ECI call is made to CICS, but control returns immediately to the Visual Basic Program, and the reply from CICS must be retrieved later using the **Poll** method on the Flow object:

```
Sub ECIDsync_Click()
    Set Connect = New CclOConn
    Set Flow = New CclOFlow
    Set Buffer = New CclOBuf
    Connect.Details "CICSNAME", "sysad", "sysad"
    Flow.SetSyncType cclDSync
    Buffer.SetString "Hello"
    Connect.Link Flow, "ECIWTO", Buffer, UOW
End Sub
```

The call to CICS is now in progress. At a later stage (in response to a user action, or perhaps when the Visual Basic program has completed some other task) the **Poll** method is used on the Flow object to collect the reply from CICS. Note that the **Poll** method requires a Buffer object as parameter if reply data is expected from CICS

```
Sub ECIReply_Click()
    If Flow.Poll(Buffer) Then
        Text1.Text = Buffer.String
    Else
        Text1.Text = "No reply from CICS yet"
    End If
End Sub
```

ECI security

You can perform security management on servers that support Password Expiry Management. Refer to the *CICS Transaction Gateway: Administration* book for your operating system, for more information on supported servers and protocols.

To use these features you must first create a connection object and invoke the **Details** method to associate a userid and password with the object. The two methods available are **Verify Password** that checks the userid and password within the connection object with the Server Security System, and **ChangePassword** that allows you to change the password at the server. If successful the connection object password is updated accordingly.

If either call is successful, you are returned a CclOSecAttr object. This object provides access to information such as last verified time, expiry time and last access time. If, for example, you query the last verified time, you are returned a CclOSecTime object and you may use the SecTime COM class methods to obtain the information in various formats. The following code shows the use of these various objects.

```
 ' Connection object already created called conn

on error goto pemhandler

Dim SecAttr as Ccl0SecAttr
Dim LastVerified as Ccl0SecTime
Dim lvdate as Date

Set SecAttr = conn.VerifySecurity
Set LastVerified = SecAttr.LastVerifiedTime

lvdate = LastVerified.GetDate
strout = Format(lvdate, "hh:mm:ss, dddd, mmm d yyyy")
Text1.Text = strout

exit sub

pemhandler:

' handle a expired password here maybe
end sub
```

ECI CICS Server Information and Connection Status

The **ECI** COM class provides the names and descriptions of CICS servers configured in the Gateway initialization file. The **Connect** COM class provides methods for querying the availability of a particular CICS server.

Object variables are declared as before, this time we use **ECI**, **Connect** and **Flow** COM classes:

```
  'Declare object variables
  Dim ECI As Ccl0ECI
  Dim Connect As Ccl0Conn
  Dim Flow As Ccl0Flow
```

On user request, the objects are created, and a list of CICS server names and their descriptions is constructed:

```
Sub ECIServers_Click()
    Dim I as Integer

    'Instantiate CICS ECI objects
    Set ECI = New Ccl0ECI
    Set Connect = New Ccl0Conn
    Set Flow = New Ccl0Flow

    'List CICS server information
    For I = 1 To ECI.ServerCount
        List1.AddItem ECI.ServerName(I)
        List1.AddItem ECI.ServerDesc(I)
    Next
 End Sub
```

A synchronous status call to the first server is made, and the results of the call displayed in a text field:

```
    Connect.Details ECI.ServerName(1)
    Connect.Status Flow
    Text1.Text = Connect.ServerStatusText
```

ECI COM classes

Table 23 lists the ECI COM classes that the CICS COM servers provide. Details of the methods these provide are in *CICS Transaction Gateway: Programming Reference*.

Table 23. ECI COM classes

COM class	Description
Buffer	Buffer used for passing data to and from a CICS server
Connection	Controls a connection to a CICS server
ECI	Provides access to a list of CICS servers configured in the Client daemon
Flow	Controls a single interaction with CICS server program
SecAttr	Provides information about security attributes (passwords)
SecTime	Provides date and time information
UOW	Coordinates a recoverable set of calls to a CICS server

Making External Presentation Interface Calls from a COM Client Program

Adding a terminal to CICS

Adding a terminal to CICS using Visual Basic

Note: These classes do not contain any specific support for 3270 data streams that contain DBCS fields. Data streams with a mixture of DBCS and SBCS fields are not supported.

The first step is to declare object variables for the EPI interfaces to be used, usually in the General Declarations section of a Visual Basic program:

```
Dim EPI As CclOEPI
Dim Terminal As CclOTerminal
Dim Session As CclOSession
Dim Screen As CclOScreen
Dim Field As CclOField
```

The required EPI objects are then instantiated using the Visual Basic **New** function. This can be done in the **Form_Load** subroutine or at a later stage in response to a user action.

The CclOEPI object must be created first to initialize the Client daemon EPI. A CclOTerminal object can then be created, and a connection established to a specific CICS server using the **Terminal.Connect** method. The first parameter to this method is the CICS server name (as configured in the Gateway initialization file), the other parameters specify additional connection details. See *CICS Transaction Gateway: Programming Reference* for additional information.

```
Sub EPIConnect_Click()
    'Create Ccl.EPI first to initialize EPI
    Set EPI = New CclOEPI
    'Create a terminal object and connect to CICS
    Set Terminal = New CclOTerminal
    Terminal.Connect "CICSNAME","",""
    'Create a session object (defaults to synchronous)
    Set Session = New CclOSession
End Sub
```

Adding a terminal to CICS using VBScript

This is again very similar to Visual Basic but differs in how you create the objects. You do not need to have to Dim your variables but it is good coding practice to do so. As with Visual basic COM objects do not support DBCS fields in the 3270 data streams. To create objects you must use the **CreateObject function**, for example:

```
Sub EPIConnect_Click()
    ' Create Ccl.EPI first to initialise EPI
    Set EPI = CreateObject("Ccl.EPI")
    ' Create a terminal object and connect to CICS
    Set Terminal = CreateObject("Ccl.Terminal")
   Terminal.Connect "CICSNAME","",""
   ' Create a session object (defaults to synchronous)
   Set Session = CreateObject("Ccl.Session")
 End Sub
```

In a similar manner, to create a Map object you issue

```
Set Map = CreateObject("Ccl.MAP")
```

Screen objects and Fields Objects are created for you.

Sending and receiving data

Having connected a CclOTerminal object to the required CICS server the **Terminal**, **Session**, **Screen** and **Field** COM classes are used to start a transaction on CICS and navigate through 3270 panels, accessing 3270 fields as required by the application.

The required CICS transaction is started using its four character transaction code. Initial transaction data can also be supplied on the **Terminal.Start** method, in this example no data is required. To access the 3270 data returned by CICS, a screen object is obtained from the terminal object, and a variety of methods can be used to obtain fields from the screen and read and update text and attributes in the fields:

```
Sub EPIStart_Click()
    'Start CESN transaction
    Terminal.Start Session, "CESN", ""
    'Get the screen object
    Set Screen = Terminal.Screen
    'Output the text from some 3270 fields
    Set Field = Screen.FieldByIndex(5)
    List1.AddItem Field.Text
    Set Field = Screen.FieldByIndex(6)
    List1.AddItem Field.Text
```

The CESN transaction is waiting for input from the user, the program could enter text into some fields and continue the transaction, in this example we simply end the transaction by sending PF3 to CICS.

```
        'Send PF3 back to CICS to end CESN
        Screen.SetAID cclPF3
        Terminal.Send Session
        'Output the text from a 3270 field
        Set Field = Screen.FieldByIndex(1)
        List1.AddItem Field.Text
    End Sub
```

Finally, disconnect the terminal, and then terminate the EPI. After you have disconnected the terminal it is recommended that you set Session, Terminal and EPI to *Nothing*. Disconnect the terminal before setting these objects; you cannot disconnect a terminal that you have set to Nothing.

```
Sub EPIDone_Click()
    Terminal.Disconnect
    'Delete the EPI COM objects
    Set Field = Nothing
    Set Screen = Nothing
    Set Session = Nothing
    Set Terminal = Nothing
    Set EPI = Nothing
End Sub
```

EPI call synchronization types

The EPI COM classes support synchronous (“blocking”) and deferred synchronous (“polling”) protocols. The Visual Basic environment does not support the asynchronous calls that are available in the C++ classes.

In the previous example a Session object was used with the default synchronization type of cclSync. When this Session object was used as the first parameter on **Terminal.Start** or **Terminal.Send**, a synchronous link call was made to CICS. The Visual Basic program was then blocked until the reply was received from CICS. When the call returned updated screen data from CICS was immediately available in the Screen object.

To make a deferred synchronous call you use the **Session.SetSyncType** method to set the Session to cclDSync. When this Session object is used on a **Terminal.Start** or **Terminal.Send** call, the screen contents are transmitted to CICS as 3270 data stream, but the method returns immediately. This allows the Visual Basic program to continue other tasks, including user interactions, while the CICS server transaction is running. Further 3270 screen updates from CICS must be retrieved later using the **Poll** method on the Terminal object:

```
Sub EPIDSync_Click()
    'Create a session object (deferred synchronous)
    Set Session = New CclOSession
    Session.SetSyncType cclDSync
    Terminal.Start Session, "CESN", ""
End Sub
```

The transaction is now in progress in the CICS server. At a later stage (in response to a user action, or when the Visual Basic program has completed some other task) the **Terminal.PollForReply** method is used to collect the reply from CICS:

```
Sub EPIReply_Click()
  If terminal.State <> cclDiscon And terminal.State <> cclError Then
    If terminal.PollForReply Then
      'Screen has been updated, output some fields
      Set Screen = Terminal.Screen
      Set Field = Screen.FieldByIndex(1)
      List1.AddItem Field.Text
    Else
      List1.AddItem "No Reply from CICS yet"
    End If
  End If
End Sub
```

A CICS server transaction may send more than one reply in response to a **Terminal.Start** or **Terminal.Send** call. More than one **Terminal.PollForReply** call may therefore be needed to collect all the replies. Use the **Terminal.State** method to find out if further replies are expected. If there are, the value returned will be cclServer.

Converting BMS maps and using the Map class

Many CICS server programs use Basic Mapping Support (BMS) to implement their 3270 screen designs. The server programs can then use symbolic names for the individual screen maps and for the 3270 fields on those maps. If the BMS source files are available, they can be copied to the Client daemon development environment and used in the implementation of a Visual Basic EPI program.

The CICS BMS Conversion Utility (`CICSBMSC.EXE`) that is provided produces a Visual Basic definitions file (a `.BAS` file) from the source BMS file (`.BMS` file). This definitions file can then be included in a Visual Basic program, and the same symbolic names used to identify maps and their fields in the server program can be used in the client program with the EPI **Map** COM class.

The `/B` option should be specified when running the conversion utility to produce Visual Basic definitions:

```
CICSBMSC /B <filename>.BMS
```

The following example shows how to use the **Map** COM class to access fields by their BMS symbolic names:

```
Dim EPI As Ccl0EPI
Dim Terminal As Ccl0Terminal
Dim Session As Ccl0Session
Dim Screen As Ccl0Screen
Dim Map as Ccl0Map
Dim Field As Ccl0Field
```

First the EPI is initialized and a 3270 terminal connection to CICS is started as in the earlier example:

```
Sub EPIConnect_Click()
    'Create Ccl.EPI first to initialize EPI
    Set EPI = New Ccl0EPI
    'Create a terminal object and connect to CICS
    Set Terminal = New Ccl0Terminal
    Terminal.Connect "CICSNAME","",""
    'Create a session object (defaults to synchronous)
    Set Session = New Ccl0Session
End Sub
```

Then the BMS application is started. This example uses a transaction code "EPIC" which runs the CICS supplied server program `EPIINQ`:

```
Sub EPIRunBMS_Click()
    Terminal.Start Session, "EPIC", ""
    Set Screen = Terminal.Screen
```

At this point the CICS server program has returned the first screen to the client. This is expected to be a known map "MAPINQ1" so we create a Map object, and use the **Map.Validate** method to initialize it and to verify that we received the expected 3270 screen. Fields can then be accessed using the **Map.FieldByName** method:

```
    Set Map = New Ccl0Map
    If (Map.Validate(Screen,MAPINQ1)) Then
        Set Field = Map.FieldByName(MAPINQ1_PRODNAM)
        List1.AddItem Field.Text
        Set Field = Map.FieldByName(MAPINQ1_TIME)
        List1.AddItem Field.Text
    Else
        List1.Text= "Unexpected screen data"
    End If
```

A more complex application would then enter data into selected fields, set the required AID key (Enter, Clear, PF or PA key) and navigate through further screens as required. The client application can mix the use of the **Screen** COM class (and its **FieldByIndex** and **FieldByPosition** methods) with the use of the **Map** COM class.

Support for Automatic Transaction Initiation (ATI)

Client applications can control whether ATI transactions are allowed by using the setATI and queryATI methods on the Terminal COM class. The default setting is for ATIs to be disabled. The following code fragment shows how to enable ATIs for a particular terminal:

```
// Create terminal connection to CICS server
Dim terminal as CclOTerminal
Set terminal = new CclOTerminal
terminal.details "MYSERVER","",""
terminal.setATI CclATIEnabled
```

The Ccl Terminal class runs any outstanding ATIs as soon as a transaction ends, and calls Additional programming needed to handle the ATI replies, and to run ATIs before or between client-initiated transactions, depending on the call synchronization type used:

Synchronous

When you call the Terminal send method, any outstanding ATIs are run after the client-initiated transaction has completed. The Terminal class waits for the ATI replies then updates the CclOScreen object contents as part of the synchronous send call. If you expect an ATI to occur before or between client-initiated transactions, call the Ccl Terminal receiveATI method to wait synchronously for the ATI.

Deferred synchronous

After the CclTerminal Start or Send method is called for a deferred synchronous session, the Poll or PollForReply method is used to receive the replies. Outstanding ATIs are started when the last reply is received (that is on the final Poll or PollForReply method). You can also call the Poll or PollForReply method to start and receive replies for ATIs between client-initiated transactions.

As the Poll or PollForReply methods can be called before or between client-initiated transactions, the receiveATI method is not needed (and is invalid) for deferred synchronous sessions.

EPI Security

You can perform security management on servers that support Password Expiry Management. Refer to the *CICS Transaction Gateway: Administration* book for your operating system, for more information on supported servers and protocols.

To use these features you first must have created a Terminal object and invoked the **SetTerminalDefinition** method to associate a userid and password with the object. The two methods available are VerifyPassword which checks the userid and password within the terminal object with the Server Security System, and ChangePassword which allows you to change the password at the server. If successful, the terminal object password is updated accordingly.

If either call is successful, you are returned a CclOSecAttr object. This object provides access to information such as last verified Date and Time, Expiry Date and Time and Last access Date and Time. If you query for example last verified Date, you are returned a CclOSecTime object which allows you to get the information in various formats. The following shows the use of these various objects.

```
' Terminal object already created called term

on error goto pemhandler

dim SecAttr as CclOSecAttr
dim LastVerified as CclOSecTime
dim lvdate as Date

set SecAttr = term.VerifyPassword
set LastVerified = SecAttr.LastVerifiedTime

lvdate = LastVerified.GetDate
strout = Format(lvdate, "hh:mm:ss, dddd, mmm d yyyy")
Text1.Text = strout

 exit sub

 pemhandler:
' handle a expired password here maybe

 end sub
```

EPI CICS Server Information

The **EPI** COM class provides the names and descriptions of CICS servers configured in the Gateway initialization file.

An EPI object is created as in the previous examples, and a list of CICS server names and their descriptions is output to a listbox “List1”:

```
Sub EPIServers_Click()
    Dim I
    'Instantiate CICS EPI object
    Set EPI = New CclOEPI
    'List CICS server information
    For I = 1 To EPI.ServerCount
        List1.AddItem EPI.ServerName(I)
        List1.AddItem EPI.ServerDesc(I)
    Next
```

EPI COM classes

Table 24 on page 170 lists the EPI COM classes that the CICS COM servers provide. Details of the methods these provide are in *CICS Transaction Gateway: Programming Reference*.

Table 24. EPI COM classes

COM class	Description
EPI	Initializes and terminates the CICS EPI and provides access to a list of CICS servers configured in the Client daemon
Field	Provides access to a single 3270 field on a screen.
Map	Provides access to 3270 fields defined by a CICS server BMS map
Screen	Provides access to a 3270 terminal screen
Session	Controls a sequence of 3270 terminal interactions with a CICS server
Terminal	Controls a 3270 terminal connection

Problem determination for COM Client programs

Handling Exceptions

With the ECI and EPI classes there appear to be two ways to check for problems when invoking methods.

One way could be to use the ErrorWindow method and set it to false, then check the ExCode and ExCodeText methods after a call to see what the return codes are. This is ***not*** the recommended way to do it and only exists now to support compatibility with earlier versions for old applications.

The recommended way is to use the Err objects which Visual Basic and VBScript provide. An Err object contains the information about an error. Visual Basic supports `On Error Goto` and `On Error Resume` features to detect that an error has occurred. VBScript only supports the `On Error Resume Next` feature. If you use `On Error Resume Next` either in Visual Basic or VBScript, you must always enter this line before any COM object call that you expect could return an error. Visual Basic/VBScript might not reset the Err variable unless you do this.

The type of interface you have selected (you DIM'ed a variable as either Object or classname) will affect the value contained in the Err.number property. It is possible to write a generic routine that handles all values in Err.Number and converts them to the documented ExCode error codes available. The example code following shows how to achieve this.

To get full advantage of this technique, ensure that you get full information in the Err object. Issue the following call after creating the ECI or EPI object:

```
ECI.SetErrorFormat 1
```

or, for EPI:

```
EPI.SetErrorFormat 1
```

Figure 15 shows how to handle errors in Visual Basic.

```
Private Sub Command1_Click()
'
' The following code assumes you have created the
' required objects first, ECI, Connect, Flow, UOW,
' Buffer
'
On Error GoTo ErrorHandler
conn.Link flow, "EC01", buf, uow
Exit Sub
ErrorHandler:
'
' Ok, the Connect call failed
' Parse the Error Number, this will work regardless of
' how the ECI objects were Dimmed
'
Dim RealError As CclECIExceptionCodes
RealError = (Err.Number And 65535) - eci.ErrorOffset

If RealError = cclTransaction Then
'
' Transaction abend, so query the Abend code
'
  AbendCode = flow.AbendCode
  If AbendCode = "AEY7" Then
    MsgBox "Invalid Userid/Password to execute CICS Program", , "CICS ECI Error"
  Else
    MsgBox "Unable to execute EC01, transaction abend:" + AbendCode, , "CICS ECI Error"
  End If
Else
   MsgBox Err.Description, , "CICS ECI Error"
End If
End Sub
```

Figure 15. Visual Basic exception handling sample

Figure 16 on page 172 shows error handling code for VBScript.

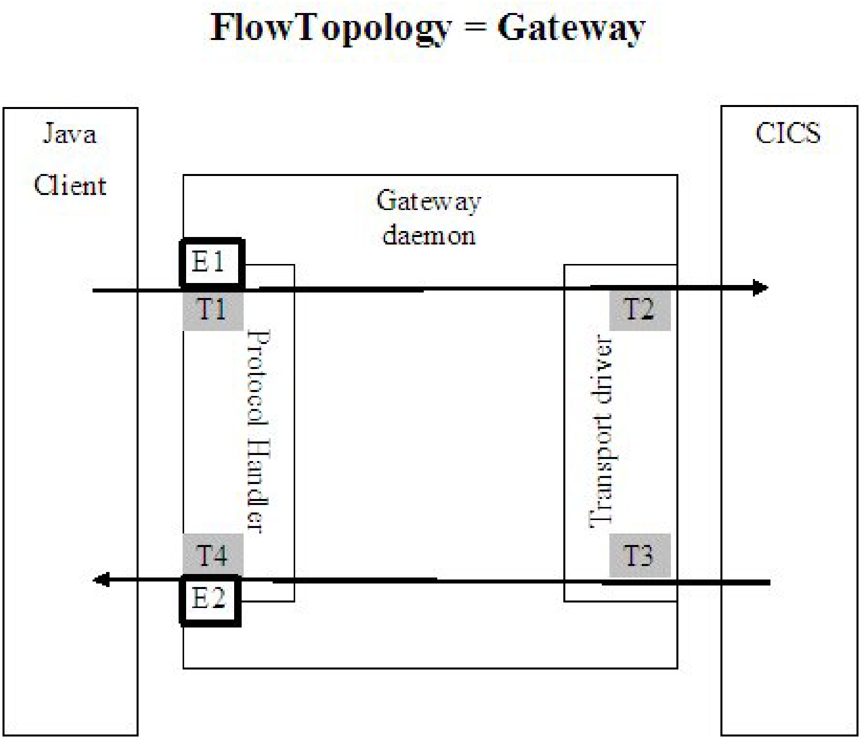
```
On Error Resume Next
con.Link flow, "EC01", buf, uow
if Err.Number <> 0 then
'
' Ok, the Connect call failed
' Parse the Error Number, this will work regardless of
' how the ECI objects were Dimmed
'
  RealError = Err.Number And 65535 - eci.ErrorOffset
'
' 13 = CclTransaction, a transaction abend.
'
  If RealError = 13 Then
'
' Transaction abend, so query the Abend code
'
    AbendCode = flow.AbendCode
    If AbendCode = "AEY7" Then
      Wscript.Echo "Invalid Userid/Password to execute CICS Program"
    Else
      Wscript.Echo "Unable to execute EC01, transaction abend:", AbendCode
    End If
   Else
     Wscript.Echo Err.Description
   End If
 End If
```

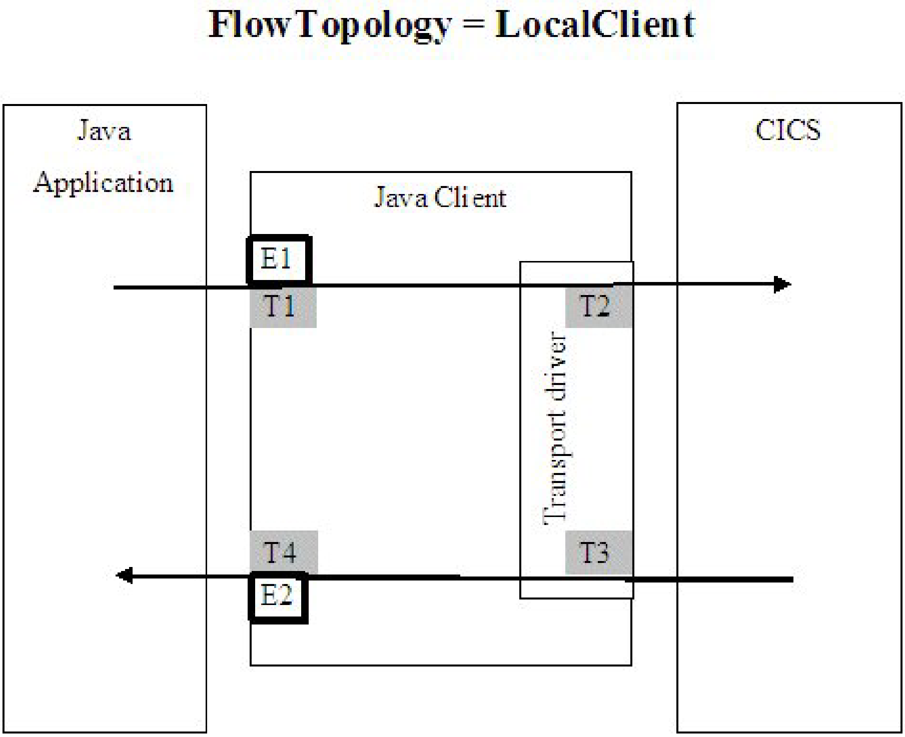
Figure 16. VBScript exception handling sample

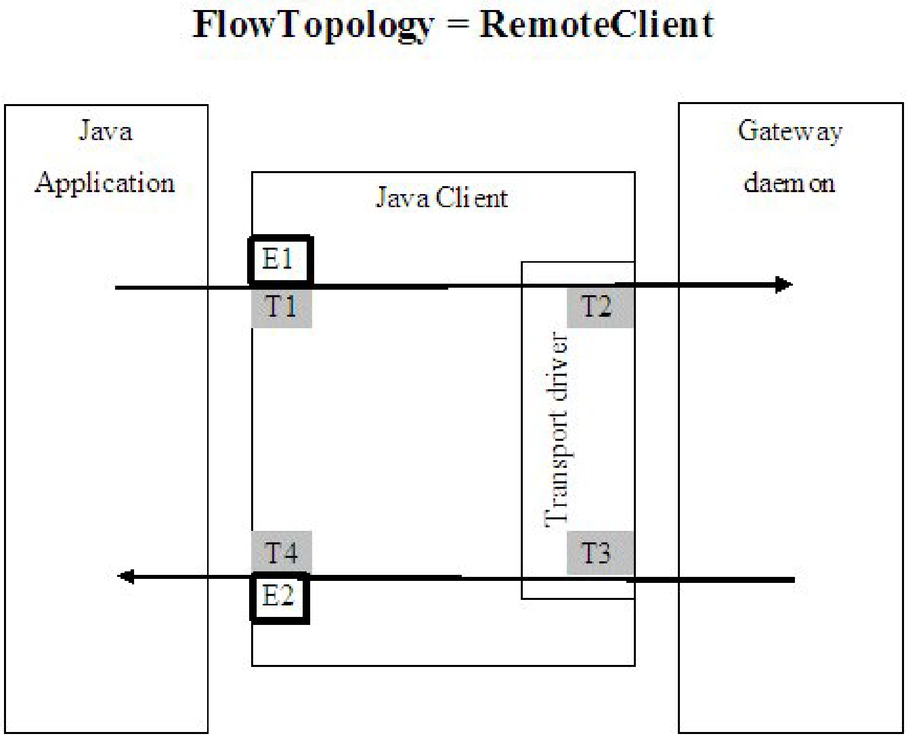
Chapter 11. Request monitoring user exits

Request level monitoring allows a third party application to be called at significant points in the request flow through the Gateway daemon and Gateway classes.

The following diagrams show where the request monitoring user exits are driven depending on the CICS Transaction Gateway configuration. In each diagram points E1 and E2 show where the exits are driven, and points T1, T2, T3 and T4 show where time stamps are collected for each request.




FlowTopology = LocalClient
Java Application
Java Client
CICS
E1
T1
T2
Transport driver
T4
T3
E2



The following set of rules apply when writing request monitoring user exits:

- Exits can be configured to run on the Gateway classes and on the Gateway daemon independently.
- Multiple exits can be configured to be active at the same time. There is no defined order in which multiple exits are called.
- Configured exits are loaded on start up and remain active for the life of the JavaGateway object or Gateway daemon.
- Exits run in-line, so should be coded to have minimum impact on performance.
- Exits that throw any exceptions or run-time errors are disabled.
- Exits are called for each ECI flow at request entry and response exit.
- Exits are called at shutdown to allow them to release resources and to end cleanly.
- At call time, exits are aware which exit point is driving the request. Data is provided to the exit to aid monitoring of the requests.

- All implementations of CICS Transaction Gateway RequestExit monitoring classes must implement the RequestExit interface.
- The default constructor can be used to set up any external resources required by the exits. For example, the sample class com.ibm.ctg.samples.requestexit.ThreadedMonitor creates a background thread to reduce the overhead for each monitored request.

Writing a monitoring application to use the exits

A RequestExit object is defined by a class that implements the RequestExit interface. At runtime a single RequestExit object is created for each configured request level monitor. Each object receives eventFired() method calls at the start of the request (E1) and at the end of the reply (E2) for each flow. These are shown by E1 and E2 on the diagrams. Timestamps are taken during the flow at T1, T2, T3 and T4 on the diagrams.

- Timestamp T1 (RequestReceived) is generated as a request arrives at the Gateway daemon or Gateway classes. This data is available when the request event type is RequestEntry or ResponseExit.
- Timestamp T2 (RequestSent) is generated as the request leaves the Gateway daemon or Gateway classes. This data is available when the request event type is ResponseExit.
- Timestamp T3 (ResponseReceived) is generated when the reply arrives back in the Gateway daemon or Gateway classes. This data is available when the request event type is ResponseExit.
- Timestamp T4 (ResponseSent) is generated when the reply leaves the Gateway daemon or Gateway classes. This data is available when the request event type is ResponseExit.

The RequestExit object exists for the lifetime of the Gateway daemon or Gateway classes, or until it throws an exception or run-time error. When the exit is triggered the eventFired() method is called and runs on the same thread as the caller. When the eventFired() method returns, the thread continues running as before. Processing performed by the exit on this thread will impact performance and should be kept to a minimum. An example exit (com.ibm.ctg.samples.requestexit.ThreadedMonitor.java) is provided to show how to transfer this processing to a separate thread to reduce the impact on performance.

Controlling request monitoring user exits dynamically

On distributed platforms, you can use ctgadmin action “rmexit” with option “command” to send systems management commands to your request monitoring user exits. This enables you to interact with the request monitoring user exits to perform tasks like dynamically starting or stopping a particular user exit.

When you issue a systems management command with a RequestEvent of “Command”, the eventFired() method is driven for all request monitoring user exits that are active on the Gateway daemon. The input data is formed of a single entry in the map, with RequestData key ″CommandData″. The value associated with this key is a string representing the data provided via the systems management command.

Sample request monitoring user exits

A simple request monitoring user exit implementation of the RequestExit interface is in the com.ibm.ctg.samples.requestexit.StdoutMonitor class. The source code for request monitoring user exits samples is located in \samples\java\com\ibm\ctg\samples\requestexit.

Related information

Request monitoring user exit API information

Correlation points available in the exits

Correlation points are available to tie the flow data available in the exits between the exits and between flows.

For all flows, the FlowType enumeration is available. The enumeration defines the type of flow and has methods to determine other key qualities about this flow. FlowTopology can be used to distinguish between Gateway daemon flows and flows in the Gateway classes, in both local and remote mode. There is no access to the underlying ECIRequest object from the exits.

Flow correlators

Individual flows through the Gateway daemon or Gateway classes have a CtgCorrelator. This correlator is a Java int, increments for each flow, available at all RequestEvents: RequestEntry to ResponseExit, and wrapped from Integer.MaxValue (values from -2,147,483,648 to 2,147,483,647). Each Gateway daemon or JavaGateway object uses independent correlators.

The Gateway daemon or Client application’s JavaGateway object can be identified if the APPLID and APPLID Qualifier are defined and are available as CtgApplid and CtgApplidQualifier. These are Java Strings containing 1 to 8 characters.

In three tier (or remote mode) topologies, the CtgCorrelator, CtgApplid and CtgApplidQualifier of the Client application flow are available in the exits in the Gateway daemon as ClientCtgCorrelator, ClientCtgApplid and ClientCtgApplidQualifier respectively.

For IPIC transactions the origin data is available to associate the flow from a Java application through to a CICS region.

For EXCI synconreturn flows from the Gateway daemon the CtgCorrelator, CtgApplid and CtgApplidQualifier is passed to CICS as a LU6.2 style UOWID. The format of this is a byte array.

CICS Network UOWID is a byte array used by CICS to uniquely identify a unit of work. The encoding is binary for the integers and EBCDIC for the characters. The format is shown in the following table.

Table 25. Format of CICS Network UOWID

Offset	Length	Description
0	1	Length of UOWID
1	1	Length of Network ID
2	n = 3 to 17	Network ID
2	q = 1 to 8	Network qualifier
2+q	1	Dot = "."
3+q	u = 1 to 8	UOWID
3+q+u	6	Instance integer
3+q+u	4	CICS Transaction Gateway Correlator
7+q+u	2	0
9+q+u	2	Sync point count (set to 0)

Access to any user correlation data in the COMMAREA is through the PayLoad object, which is read only, and only available during the eventFired() method.

Transaction correlators

For XA transactions, the XID is available, and for EXCI transactions, where the XID is unknown to CICS, the RRMS URID is also available as the Urid object.

For extended mode ECI transactions the LUW token is available after it has been set. For example, on all exits except the RequestEntry of the first request of the transaction.

Data available by FlowType and RequestEvent

For RequestEvent types of RequestEntry and ResponseExit, data is available from several fields.

The following tables cover the data available for each FlowType. In each table, Y indicates that the field data is available for a specific flow type, N indicates that the field data is not available for the specific flow type.

Table 26. Data for non XA flows at RequestEvent = RequestEntry

Flow Type	**EciStatus**	**EciSynconreturn**	**ExtendedModeEci**	**ExtendedModeCommit**	**ExtendedModeRollback**
RequestReceived	Y	Y	Y	Y	Y
RequestSent [4]	N	N	N	N	N
ResponseReceived [4]	N	N	N	N	N
ResponseSent	N	N	N	N	N
WorkerWaitTime [2]	N	N	N	N	N
Program	N	Y	Y	N	N
TranName TpnName [5]	N	Y	Y	Y	Y
Userid	N	Y	Y	Y	Y
Server	Y	Y	Y	Y	Y
GatewayUr l[6]	Y	Y	Y	Y	Y
ClientLocation [2]	Y	Y	Y	Y	Y
Location [7]	Y	Y	Y	Y	Y
LUW Token	N	N	Y	Y	Y
FlowTopology	Y	Y	Y	Y	Y
OriginData [3]	N	N	N	N	N
PayLoad	N	Y	Y	Y	Y
WireSize [2]	Y	Y	Y	Y	Y
Urid [1]	N	N	N	N	N
CtgCorrelator	Y	Y	Y	Y	Y
FlowType	Y	Y	Y	Y	Y
CtgApplid	Y	Y	Y	Y	Y
CtgApplidQualifier	Y	Y	Y	Y	Y
ClientCtgCorrelator [8]	Y	Y	Y	Y	Y
ClientCtgApplid [8]	Y	Y	Y	Y	Y
ClientCtgApplidQualifier [8]	Y	Y	Y	Y	Y
CtgReturnCode	N	N	N	N	N
CicsReturnCode	Y	N	N	N	N
CicsAbendCode	N	N	N	N	N

Table 27. Data for non XA flows at RequestEvent = ResponseExit

Flow Type	**EciStatus**	**EciSynconreturn**	**ExtendedModeEci**	**ExtendedModeCommit**	**ExtendedModeRollback**
RequestReceived	Y	Y	Y	Y	Y
RequestSent [4]	Y	Y	Y	Y	Y
ResponseReceived [4]	Y	Y	Y	Y	Y
ResponseSent	Y	Y	Y	Y	Y
WorkerWaitTime [2]	Y	Y	Y	Y	Y
Program	N	Y	Y	N	N
TranName TpnName [5]	N	Y	Y	Y	Y
Userid	N	Y	Y	Y	Y
Server	Y	Y	Y	Y	Y
GatewayUrl [6]	Y	Y	Y	Y	Y
ClientLocation [2]	Y	Y	Y	Y	Y
Location [7]	Y	Y	Y	Y	Y
LUW Token	N	N	Y	Y	Y
FlowTopology	Y	Y	Y	Y	Y
OriginData [3]	N	Y	Y	Y	Y
PayLoad	Y	Y	Y	N	N
WireSize [2]	Y	Y	Y	Y	Y
Urid [1]	N	N	N	N	N
CtgCorrelator	Y	Y	Y	Y	Y
FlowType	Y	Y	Y	Y	Y
CtgApplid	Y	Y	Y	Y	Y

Table 27. Data for non XA flows at RequestEvent = ResponseExit (continued)

Flow Type	EciStatus	EciSynconreturn	ExtendedModeEci	ExtendedModeCommit	ExtendedModeRollback
CtgApplidQualifier	Y	Y	Y	Y	Y
ClientCtgCorrelator [8]	Y	Y	Y	Y	Y
ClientCtgApplid [8]	Y	Y	Y	Y	Y
ClientCtgApplidQualifier [8]	Y	Y	Y	Y	Y
CtgReturnCode	Y	Y	Y	Y	Y
CicsReturnCode	N	Y	Y	N	N
CicsAbendCode	N	Y	Y	N	N

Table 28. Data for XA flows at RequestEvent = RequestEntry

Flow Type	XaStart	XaEci	Xa1PhaseCommit	XaPrepare	XaCommit	XaRollback	XaForget	XaRecover
RequestReceived	Y	Y	Y	Y	Y	Y	Y	Y
RequestSent [4]	N	N	N	N	N	N	N	N
ResponseReceived [4]	N	N	N	N	N	N	N	N
ResponseSent	N	N	N	N	N	N	N	N
WorkerWaitTime [2]	N	N	N	N	N	N	N	N
Program	N	N	N	N	N	N	N	N
TranName TpnName [5]	N	Y	N	N	N	N	N	N
Userid	Y	Y	Y	Y	Y	Y	Y	Y
Server	Y	Y	Y	Y	Y	Y	Y	Y
GatewayUr l[6]	Y	Y	Y	Y	Y	Y	Y	Y
ClientLocation [2]	Y	Y	Y	Y	Y	Y	Y	Y
Location [7]	Y	Y	Y	Y	Y	Y	Y	Y
LUW Token	N	N	N	N	N	N	N	N
FlowTopology	Y	Y	Y	Y	Y	Y	Y	Y
OriginData [3]	N	N	N	N	N	N	N	N
PayLoad	N	Y	N	N	N	N	N	N
WireSize [2]	Y	Y	Y	Y	Y	Y	Y	Y
Xid	Y	Y	Y	Y	Y	Y	Y	N
Urid [1]	N	N	N	N	N	N	N	N
CtgCorrelator	Y	Y	Y	Y	Y	Y	Y	Y
FlowType	Y	Y	Y	Y	Y	Y	Y	Y
CtgApplid	Y	Y	Y	Y	Y	Y	Y	Y
CtgApplidQualifier	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgCorrelator [8]	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgApplid [8]	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgApplidQualifier [8]	Y	Y	Y	Y	Y	Y	Y	Y
CtgReturnCode	N	N	N	N	N	N	N	N
CicsReturnCode	N	N	N	N	N	N	N	N
CicsAbendCode	N	N	N	N	N	N	N	N
XaReturnCode	N	N	N	N	N	N	N	N

Table 29. Data for XA flows at RequestEvent = ResponseExit

Flow Type	XaStart	XaEci	Xa1PhaseCommit	XaPrepare	XaCommit	XaRollback	XaForget	XaRecover
RequestReceived	Y	Y	Y	Y	Y	Y	Y	Y
RequestSent [4]	Y	Y	Y	Y	Y	Y	Y	Y
ResponseReceived [4]	Y	Y	Y	Y	Y	Y	Y	Y
ResponseSent	Y	Y	Y	Y	Y	Y	Y	Y
WorkerWaitTime [2]	Y	Y	Y	Y	Y	Y	Y	Y
Program	N	Y	N	N	N	N	N	N
TranName TpnName [5]	N	Y	N	N	N	N	N	N
Userid	Y	Y	Y	Y	Y	Y	Y	Y
Server	Y	Y	Y	Y	Y	Y	Y	Y
GatewayUr l[6]	Y	Y	Y	Y	Y	Y	Y	Y
ClientLocation [2]	Y	Y	Y	Y	Y	Y	Y	Y
Location [7]	Y	Y	Y	Y	Y	Y	Y	Y

Table 29. Data for XA flows at RequestEvent = ResponseExit (continued)

Flow Type	XaStart	XaEci	Xa1PhaseCommit	XaPrepare	XaCommit	XaRollback	XaForget	XaRecover
LUW Token	N	N	N	N	N	N	N	N
FlowTopology	Y	Y	Y	Y	Y	Y	Y	Y
OriginData [3]	Y	Y	Y	Y	Y	Y	Y	Y
PayLoad	N	Y	N	N	N	N	N	N
WireSize [2]	Y	Y	Y	Y	Y	Y	Y	Y
Xid	Y	Y	Y	Y	Y	Y	Y	N
Urid [1]	Y	N	N	N	N	N	N	N
CtgCorrelator	Y	Y	Y	Y	Y	Y	Y	Y
FlowType	Y	Y	Y	Y	Y	Y	Y	Y
CtgApplid	Y	Y	Y	Y	Y	Y	Y	Y
CtgApplidQualifier	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgCorrelato [8]	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgApplid [8]	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgApplidQualifier [8]	Y	Y	Y	Y	Y	Y	Y	Y
CtgReturnCode	Y	Y	Y	Y	Y	Y	Y	Y
CicsReturnCode	N	Y	N	N	N	N	N	N
CicsAbendCode	N	Y	N	N	N	N	N	N
XaReturnCode	Y	N	Y	Y	Y	Y	Y	Y

Note: [1] Urid is only available on non-IPIC flows.

Note: [2] ClientLocation, WorkerWaitTime and WireSize are only available when FlowTopology=Gateway.

Note: [3] OriginData is only available for IPIC flows to CICS servers when FlowTopology=Gateway.

Note: [4] The timestamps from and to another system are only set if the flow goes to another system. For EciStatus and for non-IPIC XA flows, except XaEci, this will only be when FlowTopology=RemoteClient.

Note: [5] TranName and TpnName are mutually exclusive. Either might be set, but not both.

Note: [6] GatewayUrl is only available when FlowTopology=RemoteClient or FlowTopology=LocalClient.

Note: [7] Location is only available for FlowTopology=Gateway and FlowTopology=RemoteClient.

Note: [8] For requests originating from the Java client using classes from CICS TG V 7.1 or higher and FlowTopology=Gateway.

Chapter 12. ECI and EPI C exits

This information describes exits you can add to the ECI, EPI, and cicsterm when using the Client daemon. The exits allow you to influence the processing of certain application requests and can be used for monitoring purposes.

The exits must be coded in the C programming language.

Loading the exits

During ECI, EPI, cicsterm, and cicsprnt initialization, CICS Transaction Gateway attempts to load the objects described in Table 30 from the <install_path>\bin subdirectory, and to call the corresponding entry points.

Table 30. ECI and EPI exits

	Object name	**Entry point name**
ECI	**cicsecix**	**CICS_ECIEXITINIT**
EPI	**cicsepix**	**CICS_EPIEXITINIT**
cicsterm	**cicsepix**	**CICS_EPIEXITINIT**
cicsprnt	**cicsepix**	**CICS_EPIEXITINIT**

Each entry point is passed a single parameter, a pointer to a structure that contains a list of addresses. The initialization code of the program puts the addresses of all the exits into the structure, and then the exits are called at appropriate points in ECI, EPI, cicsterm, and cicsprnt processing. Because the exits are entered by using the addresses supplied, you may give the exits any valid names. In this book, conventional names are used for the exits.

For migration purposes, the CICS Transaction Gateway or CICS Universal Client first looks for a lower case named object, and then for an upper case named object. If the objects are not found, no exit processing occurs.

Sample exits and interface definitions

Sample user exit files are supplied in:

- <install_path>\samples\c\exits

See file <install_path>\samples\samples.txt for more details about the samples, and how to edit them for your system. To install the samples:

1. Make any required changes to the details of servers and aliases

2. to make files cicsecix and cicsepix:
   - On Windows, run ecix1mak.cmd and epix1mak.cmd
   - On UNIX and Linux operating systems, run samp.mak
3. copy cicsecix and cicsepix to the <install_path>\bin subdirectory

The following files are also supplied to help with programming the exits. Unless otherwise specified, the path to the files is <install_path>\samples\c\exits.

cicsecix.h
A header file in the <install_path>\include directory that defines:
- inputs and outputs for each ECI exit
- the format of the list of addresses for calling ECI exits
- data structures used by ECI exits
- return code values for ECI exits.

cicsepix.h
A header file in the <install_path>\include directory that defines:
- inputs and outputs for each EPI exit
- the format of the list of addresses for calling EPI exits
- data structures used by EPI exits
- return code values for EPI exits.

ecix1.c A template that you can use to write your own ECI user exits. It does not perform any actions if you compile it.

epix1.c
A template that you can use to write your own EPI user exits. It does not perform any actions if you compile it.

Writing your own user exits

Follow these rules:
- Do not make EPI or ECI calls from the exit.
- To minimize the impact on performance, keep executable code to a minimum.
- Ensure that all your exit code is re-entrant and thread-safe.
- Name the primary entry points as follows:

  ECI
  CICS_ECIEXITINIT

  EPI, cicsterm, and cicsprnt
  CICS_EPIEXITINIT

You may change the names of the actual exits; do not change the parameter lists.

- Ensure that your user exit programs contain valid entry points for all of the user exit functions, apart from the following:
  - **CICS_EciSetProgramAlias** is optional.
  - Include either **CICS_EpiTermIdExit** or **CICS_EpiTermIdInfoExit**. New DLLs should use **CICS_EpiTermIdInfoExit**.
  - Include either **CICS_EPIStartTranExit** or **CICS_EPIStartTranExtendedExit**. New DLLs should use **CICS_EPIStartTranExtendedExit**.

  If a required exit is not included, the exits will not load.

To use the ECI exits, you supply a CICS_ECIEXITINIT function in a DLL called cicsecix .dll (cicsecix.a on UNIX and Linux operating systems).

To use the EPI exits, you supply a CICS_EPIEXITINIT function in a DLL called cicsepix.dll (cicsepix.a on UNIX and Linux operating systems).

The CICS_ECIEXITINIT and CICS_EPIEXITINIT functions each set an ExitList structure to point to the addresses of all the exit functions contained in the exit object. For example, the sample CICS_EPIEXITINIT is as follows:

```
void CICSEXIT CICS_EPIEXITINIT(CICS_EpiExitList_t *ExitList)

{
    ExitList->InitializeExit       = &CICS_EpiInitializeExit;
    ExitList->TerminateExit        = &CICS_EpiTerminateExit;
    ExitList->AddTerminalExit      = &CICS_EpiAddTerminalExit;
    ExitList->StartTranExit        = &CICS_EpiStartTranExit;
    ExitList->ReplyExit            = &CICS_EpiReplyExit;
    ExitList->DelTerminalExit      = &CICS_EpiDelTerminalExit;
    ExitList->GetEventExit         = &CICS_EpiGetEventExit;
    ExitList->TranFailedExit       = &CICS_EpiTranFailedExit;
    ExitList->SystemIdExit         = &CICS_EpiSystemIdExit;
    ExitList->TermIdExit           = &CICS_EpiTermIdExit;
    ExitList->TermIdInfoExit       = &CICS_EpiTermIdInfoExit;
    ExitList->StartTranExtendedExit = &CICS_EpiStartTranExtendedExit;
}
```

As the exits are entered by using the addresses supplied, you can give them any name you want, as long as their function signature is exactly the same as the CICS_Eci* or CICS_Epi* functions.

InitializeExit is passed a version number of X’FF000000’ when driven by cicsterm or cicsprnt. This enables user programs to be able to differentiate between cicsterm and cicsprnt user exits, and EPI user exits if they wish to do so.

Diagnostic information

The Client API trace shows the input parameters to the exits immediately before they are called, and the output of the exit when the exit returns. A user exit active flag of `0` in the trace means that the exits have failed to activate, due to a missing required exit. This information is also shown in a log message. See "Writing your own user exits" on page 184 for information about required exits.

CICS tracing is not available for use within the exit.

EPI user exits

The following describes the EPI exits that are available and how they affect the EPI, cicsterm, and cicsprnt behavior is described.

CICS_EpiInitializeExit

EPI: This EPI exit does not affect the running of the calling EPI program, but it does allow the user to switch the user exits on or off for the process that calls it. It is called once per process that uses the EPI. It is called before any other EPI calls take place, and is called at the end of a successful **CICS_EpiInitialize**.

cicsterm:
This exit is called once only for each cicsterm session that is created, because each cicsterm runs in a separate process. The version number passed is X'FF000000'.

CICS_EpiTerminateExit

EPI: Called by **CICS_EpiTerminate**, this is always the last EPI call in a particular process. It does not affect the running of the calling EPI program. It is called after checking that the EPI was initialized, and that there is not an active notify thread, but just before EPI is actually terminated. The EPI exit DLL is unloaded immediately following the user exit call.

cicsterm:
Only called once during cicsterm termination.

CICS_EpiAddTerminalExit

EPI: Allows the user to select a server, change the server parameters passed to the EPI call, and refuse to add a terminal to a server. This all happens from within the EPI call. The EPI program subsequently refers to the server by an index number, therefore the program does not need to know what server it is actually connected to. If the user exit refuses to connect a server, then **CICS_EpiSystemIDExit** is not called (see below for further details). **CICS_EpiAddTerminalExit** is called after **CICS_EpiAddTerminal** or **CICS_EpiAddExTerminal** has verified that the EPI has been successfully initialized, and that there is a free terminal index. It is called before the **CICS_EpiAddTerminal** or **CICS_EpiAddExTerminal** call actually sends the terminal definition to the server.

cicsterm:

The /s or /r parameters of cicsterm allow the user to specify that the CICS Transaction Gateway can connect to:

- The first server defined in the CICS Transaction Gateway configuration file.
- A server chosen by the user from a list of available servers
- A server specified by the /s or /r parameter.

CICS_EpiAddTerminalExit receives the system name as a parameter, and can specify a different server if required, or reject the server and cause the terminal emulator to terminate. If **AddTerminalExit** rejects the install, cicsterm displays an error to the effect that the server is unavailable.

CICS_EpiSystemIdExit

EPI: Allows the user to re-select a server if a **CICS_EpiAddTerminal** or **CICS_EpiAddExTerminal** call fails. This user exit is not called if the exit itself causes the failure. If the exit returns CICS_EXIT_OK, **CICS_EpiAddTerminal** or **CICS_EpiAddExTerminal** tries to add the terminal again. The server parameters can be changed by this exit between retries.

CICS_EpiSystemIdExit can be called asynchronously or synchronously by EPI programs. **CICS_EpiSystemIdExit** can be presented with any of the following:

- A CICS_EPI_ERR_SYSTEM error, meaning the server is unknown
- A CICS_EPI_ERR_SERVER_DOWN error, meaning the server has failed
- A CICS_EPI_ERR_SECURITY error, for a security failure
- A CICS_EPI_ERR_FAILED error for any other type of failure.

It is also passed a parameter that is the same as the **cics_syserr_t** data structure **cause** field. This value further specifies the error and is a value specific to the operating environment

cicsterm:

If a cicsterm terminal add call fails due to the Client daemon not having enough sessions free, **SystemIdExit** is called with CICS_EPI_ERR_FAILED as the primary reason code and 7046 as the secondary reason code indicating a resource shortage. In all other cases of CICS_EPI_ERR_FAILED, cicsterm passes a secondary reason code of 0.

If no user exits are active, then cicsterm retries a terminal install if it fails due to there not being enough available sessions. (This allows terminals to wait for free sessions before being installed.) If there are user exits active, any retry behavior is controlled completely by the exit.

CICS_EpiTermIdExit

EPI: Allows the user to know what EPI Termid an added terminal is given. This is only called after a terminal has been successfully installed on a server. It does not affect the running of the EPI program. EPI Termid numbers are local to each process the EPI program runs under.

cicsterm:
As only one cicsterm runs per process, the Termid number is always set to 1.

CICS_EpiTermIdInfoExit

EPI: Allows the user to know details about a terminal. This is called after a terminal has been successfully installed on a server.

cicsterm:
As only one cicsterm runs per process, the Termid number is always set to 1.

CICS_EpiDelTerminalExit

EPI: Is called when **CICS_EpiDelTerminal** is issued. It does not affect the running of the EPI program.

cicsterm:
As only one cicsterm runs per process, the Termid number is always set to 1. It is called just before the **CICS_EpiTerminateExit** when the terminal is ended. When the server fails the **CICS_EpiAddTerminalExit** is called again when it is restarted. However the **CICS_EpiDelTerminalExit** is not called when the server fails.

CICS_EpiStartTranExtendedExit/CICS_EpiStartTranExit

EPI: Allows a user to see that a transaction has been started, and to see the Transid, 3270 data, and Termid (**CICS_EpiStartTranExtendedExit** only) sent to it. It does not affect the running of the EPI program. **CICS_EpiStartTranExtendedExit/CICS_EpiStartTranExit** is called after the EPI state has been verified, and just before the request to start the transaction is sent to the Client daemon.

Note that a pseudo-conversational transaction causes the exit to be called for each actual transaction.

cicsterm:
If a non-ATI transaction is being started, the exit is called, sending a blank in the Transid field and the TIOA (terminal input output area) for the Data field. As only one cicsterm runs per process, the Termid number is always set to 1. The Transid is either the first four characters of the TIOA data, or follows a 3270 Set Buffer Address (SBA) command (which begins X'11'). In the latter case, it starts on the 4th byte of the TIOA (as a SBA command takes up a total of three bytes).

CICS_EpiStartTranExtendedExit/CICS_EpiStartTranExit is not driven for ATI transactions. However pseudo-conversational transactions drive the exit. In the case of pseudo-conversational transactions, the transaction id is put in the transid parameter block and the TIOA passed in the data block does not contain the transaction id.

StartTranExtendedExit is not called as a result of an EXEC CICS RETURN TRANSID*name* IMMEDIATE command issued by an application from a cicsterm session.

CICS_EpiReplyExit

EPI: Allows the user to see when an application sends a data reply to CICS. It does not affect the running of the EPI program.

cicsterm:
Activated when the cicsterm is sending data to CICS and a transaction is currently active. The Termid number is always set to 1. The terminal TIOA is passed to **ReplyExit**.

ReplyExit is not called as a result of an EXEC CICS RETURN TRANSID*name* IMMEDIATE command issued by an application from a cicsterm session.

The **CICS_EpiGetEventExit** and **CICS_EpiTranFailedExit** exits are called only for the EPI and not for cicsterm and cicsprnt.

Java request monitoring exits

These exits are only available on the CICS Transaction Gateway.

Online programming reference information is provided for the Java classes and interfaces provided with CICS Transaction Gateway.

The reference information is in HTML format and is generated using the Javadoc tool provided with the JDK.

No CICS TG API calls must be made from the exit program.

See the README file for the latest information on using the programming reference information.

Related information

Request monitoring user exit API information

Appendix A. ECI extensions that are environment-dependent

This information describes extensions to the ECI that are supported only in certain environments.

Call type extensions

The following call types are for asynchronous calls.

For more information about the program link calls, see Table 31 on page 194, and *ECI_ASYNC call type* in the *C and Cobol* chapter of *CICS Transaction Gateway: Programming Reference*.

For more information about the status information calls, see Table 31 on page 194, and *ECI_STATE_ASYNC call type* in *CICS Transaction Gateway: Programming Reference*.

Asynchronous program link call, with notification by message (ECI_ASYNC_NOTIFY_MSG)

This call type is available only for programs running on Windows.

The calling application gets control back when the ECI accepts the request. Note that this does not indicate that the program has started to run, merely that the parameters have been validated. The request might be queued for later processing.

The ECI sends a notification message to the specified window when the response is available. (For details of the message format, see "Reply message formats" on page 194.) On receipt of this notification, the calling application should use ECI_GET_SPECIFIC_REPLY to receive the actual response. The ECI_GET_REPLY call type is deprecated.

The following fields are required parameters for notification by message:

- **eci_async_notify.window_handle**
- **eci_message_id** indicates the message type to be used in the notification process.

eci_message_qualifier can be used as an input to provide a user-defined name for the call. It is returned as part of the notification message for the Windows environment.

Asynchronous program link call, with notification by semaphore (ECI_ASYNC_NOTIFY_SEM)

This call type is available only for programs running on Windows.

The calling application gets control back when the ECI accepts the request. Note that this does not indicate that the program has started to run, merely that the parameters have been validated. The request might be queued for later processing.

The ECI posts the specified semaphore when the response is available. On receipt of this notification, the calling application should use ECI_GET_SPECIFIC_REPLY to receive the actual response.

eci_message_qualifier can be used as an input to provide a user-defined name for the call.

The following field is a required parameter for notification by semaphore:
- **eci_async_notify.sem_handle** refers to the semaphore.

Asynchronous status call, with notification by message (ECI_STATE_ASYNC_MSG)

This call type is available only for programs running on Windows.

eci_message_qualifier can be used as an input to provide a user-defined name for the call.

The ECI sends a notification message to the specified window when the response is available. (For details of the message format, see “Reply message formats” on page 194.) On receipt of this notification, the calling application should use ECI_GET_SPECIFIC_REPLY to receive the actual response.

For details of the additional parameters relating to notification by message, see the description of the ECI_ASYNC_NOTIFY_MSG call type.

Asynchronous status call, with notification by semaphore (ECI_STATE_ASYNC_SEM)

This call type is available only for programs running on Windows.

eci_message_qualifier can be used as an input to provide a user-defined name for the call.

The ECI posts the specified semaphore when the response is available. On receipt of this notification, the calling application should use ECI_GET_SPECIFIC_REPLY to receive the actual response.

The following field is a required parameter for notification by semaphore:

- **eci_async_notify.sem_handle** refers to the semaphore.

Fields to support ECI extensions

The following fields in the ECI parameter block are to support environment-dependent extensions.

eci_async_notify.window_handle
(Windows environment, ECI_ASYNC_NOTIFY_MSG and ECI_STATE_ASYNC_MSG call types)

The handle of the window to which the reply message will be posted.

The ECI uses this field as input only.

Note: eci_window_handle is a synonym for this parameter.

eci_async_notify.sem_handle
(Windows environment, ECI_ASYNC_NOTIFY_SEM and ECI_STATE_ASYNC_SEM call types)

Windows applications should pass an event object handle.

The ECI uses this field as input only.

eci_async_notify.win_fields.hwnd

The handle of the Windows window to which the reply message will be posted.

The ECI uses this field as input only.

eci_async_notify.win_fields.hinstance

The Windows hInstance of the calling program as supplied during program initialization.

The ECI uses this field as input only.

eci_sync_wait.hwnd

The handle of the window that is to be disabled during the synchronous call.

The ECI uses this field as input only.

eci_message_id
(Windows environment, ECI_ASYNC_NOTIFY_MSG and ECI_STATE_ASYNC_MSG call types)

The message identifier to be used for posting the reply message to the window specified in the relevant window handle.

The ECI uses this field as input only.

Reply message formats

When an application makes an asynchronous call requesting notification by message, the ECI returns the result in a message to a window using the specified window handle and message identifier.

The message is divided into two parameters, as follows:

wParam

High-order 16 bits
Specified message qualifier

Low-order 16 bits
Return code

lParam
4-character abend code, if applicable

ECI return notification

Table 31. CICS_ExternalCall return codes — environment-dependent extensions

Return code	Meaning
ECI_ERR_NULL_WIN_HANDLE	An asynchronous call was specified with the window handle set to 0.
ECI_ERR_NULL_MESSAGE_ID	An asynchronous call was specified with the message identifier set to 0.
ECI_ERR_NULL_SEM_HANDLE	A null semaphore handle was passed when a valid handle was required.

Summary of input parameter requirements

Table 32 on page 195 shows the input parameters for an ECI call, and, for each call type, whether the parameters are required (R), optional (O), or not applicable (-). Where a parameter is shown as optional or not-applicable an initial field setting of nulls is recommended. An asterisk (*) immediately following an R means that further details regarding applicability are given under the description of the parameter.

The following abbreviations are used in the **Parameter** column:

AN	**async_notify**
WF	**win_fields**
SW	**sync_wait**

Also, all named parameters have an **eci_** prefix. Thus **AN.WF.hwnd** represents the **eci_async_notify.win_fields.hwnd** parameter.

The following 3-character abbreviations are used for the call types in the column headings of the table:

ANM ECI_ASYNC_NOTIFY_MSG

ANS ECI_ASYNC_NOTIFY_SEM

SAM ECI_STATE_ASYNC_MSG

SAS ECI_STATE_ASYNC_SEM

SYN ECI_SYNC

SSN ECI_STATE_SYNC

Table 32. Input parameters for CICS_ExternalCall — environment-dependent extensions

Parameter, eci_	ANM	ANS	SAM	SAS	SYN	SSN
call_type	R	R	R	R	R	R
program_name	R*	R*	-	-	R*	-
userid	R	R	-	-	R	-
password	R	R	-	-	R	-
transid	O	O	-	-	O	-
commarea	O	O	R*	R*	O	R*
commarea_length	O	O	R*	R*	O	R*
timeout	O	O	O	O	O	O
extend_mode	R	R	R	R	R	R
AN.window_handle	R*	-	R*	-	-	-
AN.sem_handle	-	R	-	R	-	-
AN.WF.hwnd	R*	-	R*	-	-	-
AN.WF.hinstance	R*	-	R*	-	-	-
SW.hwnd	-	-	-	-	R*	R*
message_id	R	-	R	-	-	-
message_qualifier	O	O	O	O	O	O
luw_token	R	R	R*	R*	R	R*
version	O	O	O	O	O	O
system_name	O	O	O	O	O	O

Appendix B. Sample programs

Samples are supplied in subdirectories of the <install_path>\samples directory. See file samples.txt, in the <install_path>\samples directory, for details, including compilation instructions and information on compiler considerations.

Different levels of sample are provided. These include

- A simple sample to allow you to test that the CICS Transaction Gateway is functioning and to give you a feel for the basic requirements of a Client application.
- More complex samples that demonstrate advanced API features, and provide a more realistic example of a Client application.

As far as possible the sample code adheres to the language standards, for example, ANSI C. The samples are all driven from the command line to avoid any dependence on platform-specific GUI code.

Appendix C. Return codes from the ctgadmin command

The ctgadmin command can be invoked through a script. The return codes listed in this information are for errors that can be dealt with by the user. Return codes not listed below indicate an internal processing error, and are used by the service organization in the event of a persistent problem.

0	The command completed successfully, or help was requested
1	The product is not correctly installed
2	Failure reading registry key
3	Failure writing registry key
4	Java not found on system
11	Java Version not supported
13	The product is not correctly installed; ctgadmin.jar cannot be located
14	Operating system not supported
100	Command failed due to bad parameter specification
101	Failure communicating with the CICS Transaction Gateway
102	Attempt to connect on non admin port
104	Failure to locate messages file

The product library and related literature

This information lists books on the CICS Transaction Gateway, and related topics.

CICS Transaction Gateway books

- *CICS Transaction Gateway: Windows Administration, SC34-6960-00*

  This book describes the administration of the CICS Transaction Gateway for Windows.
- *CICS Transaction Gateway: UNIX and Linux Administration, SC34-6959-00*

  This book describes the administration of the CICS Transaction Gateway for UNIX and Linux.
- *CICS Universal Client: Windows Administration, SC34-6963-00*

  This book describes the administration of the CICS Transaction Gateway for Windows.
- *CICS Universal Client: UNIX and Linux Administration*, SC34-6962-00

  This book describes the administration of the CICS Transaction Gateway for the Linux operating system.
- *CICS Transaction Gateway: z/OS Administration, SC34-6961-00*

  This book describes the administration of the CICS Transaction Gateway for z/OS.
- *CICS Transaction Gateway: Messages*, SC34-6964-00

  This online book lists and explains the error messages that can be generated by the CICS Transaction Gateway.
- *CICS Transaction Gateway: Programming Reference*, SC34-6966-00

  This book provides information on the APIs of the programming languages supported by the CICS Transaction Gateway.

  Additional HTML pages contain JAVA programming reference information.
- *CICS Transaction Gateway: Programming Guide*, SC34-6965-00

  This introduction to programming for the CICS Transaction Gateway provides the information that you need to allow user applications to use CICS facilities in a client/server environment.

Sample configuration documents

Several sample configuration documents are available in portable document format (PDF). These documents give step-by-step guidance for configuring CICS Transaction Gateway for communication with CICS servers, using various protocols. They provide detailed instructions that extend the information in the CICS Transaction Gateway library.

Visit the following Web site:

`www.ibm.com/software/cics/ctg`

and follow the **Library** link.

Redbooks

The following International Technical Support Organization (ITSO) Redbook publication contains many examples of client/server configurations:

- *CICS Transaction Gateway V5 - The WebSphere Connector for CICS, SG24-6133*
- *Revealed! Architecting Web Access to CICS, SG24-5466*
- *Enterprise JavaBeans for z/OS and OS/390® CICS Transaction Server V2.2, SG24-6284*
- *Java Connectors for CICS: Featuring the J2EE Connector Architecture, SG24-6401*. This book provides information on developing J2EE applications.
- *Systems Programmer's Guide to Resource Recovery Services (RRS), SG24-6980-00*. This book provides information on using RRS in various scenarios.
- *Communications Server for z/OS V1R2 TCP/IP Implementation Guide, SG24-6517-00*. This book provides information on using Communications Server for z/OS V1R2, including load balancing.
- *Redpaper: Transactions in J2EE, REDP-3659-00*. This redpaper provides a discussion of transactions in the J2EE environment, including one- and XA transactions.

You can obtain ITSO Redbooks from a number of sources. For the latest information, see:

`www.ibm.com/redbooks/`

Other Useful Books

CICS Transaction Server publications

CICS Transaction Server for z/OS RACF Security Guide, SC34-6249

CICS interproduct communication

The following books describe the intercommunication facilities of the CICS server products:

- *CICS Family: Interproduct Communication*, SC34-6267
- *CICS Transaction Server for Windows V5.0 Intercommunication*, SC34-6209
- *CICS Transaction Server for z/OS CICS External Interfaces Guide*, SC34-6449
- *CICS Transaction Server for z/OS: Intercommunication Guide*, SC34-6448
- *CICS/VSE 2.3: Intercommunication Guide*, SC33-0701
- *CICS Transaction Server for iSeries V5R2: Intercommunication*, SC41-5456
- *TXSeries 5.1: CICS Intercommunication Guide*, SC09-4462

The first book above is a CICS family book containing a platform-independent overview of CICS interproduct communication.

CICS problem determination books

The following books describe the problem determination facilities of the CICS server products:

- *Transaction Server for Windows V5.0: Problem Determination*, GC34-6210
- *CICS Transaction Server for z/OS V3.1 CICS Problem Determination Guide*, SC34-6441
- *CICS/VSE 2.3 Problem Determination Guide*, SC33-0716
- *CICS Transaction Server for iSeries V5R2: Problem Determination*, SC41-5453
- *TXSeries V5.1: CICS Problem Determination Guide*, SC09-4465

You can find information on CICS products at the following Web site:

`www.ibm.com/software/cics/ctg`

Microsoft Windows publications

See this Web site:

`www.microsoft.com/windows`

APPC-related publications

IBM products

IBM Communications Server

See this Web page:

`www.ibm.com/software/network/commserver/library`

IBM Personal Communications

See this Web page:

`www.ibm.com/software/network/pcomm/library`

Microsoft products

See this page Web:

`http://www.microsoft.com/hiserver/techinfo/productdoc/default.mspx`

Systems Network Architecture (SNA)

- *SNA Formats*, GA27-3136
- *Systems Network Architecture Technical Overview*, GC30-3073
- *Guide to SNA over TCP/IP*, SC31-6527

Obtaining books from IBM

For information on books you can download, visit our Web site at:

`www.ibm.com/software/cics/ctg`

and follow the **Library** link.

Accessibility features for CICS Transaction Gateway

Accessibility features help users who have a physical disability, such as restricted mobility or limited vision, to use information technology products successfully. The CICS Transaction Gateway supports keyboard-only operation. Topics on the following pages give details of accessibility features.

Visit the IBM Accessibility Center for more information about IBM's commitment to accessibility.

Documentation

See the Eclipse information center for an HTML version of the documentation.

Starting the Gateway daemon

You can start the Gateway daemon from a command prompt using a screen reader.

In some Telnet sessions, the screen reader might reread CICS Transaction Gateway log output or the command prompt after the CICS Transaction Gateway has started. This behavior is expected, and does not mean that the CICS Transaction Gateway has failed to start.

To determine if the CICS Transaction Gateway started correctly, check for the message:

```
 'CTG6512I CICS Transaction Gateway initialization complete'.
```

If the CICS Transaction Gateway did not start successfully, this message is produced:

```
 'CTG6513E CICS Transaction Gateway failed to initialize'.
```

Setting EPITerminal properties programmatically

The EPITerminal terminal properties sheet is not accessible. To set properties programmatically, use the getTerminal() method of the EPITerminal object and cast it to a Terminal object. For example, if epiTerm is an EPITerminal object, code something like the following:

```
Terminal term = (Terminal)epiTerm.getTerminal();
```

You can then use methods on the Terminal object to set these properties. To set the name for a CICS server named `YOURSERV`, code the following:

```
term.setServerName("YOURSERV");
```

See the Javadoc supplied with the product for full details of these setter methods.

cicsterm

Although cicsterm is accessible, it relies on the application that is being processed to define an accessible 3270 screen.

The bottom row of cicsterm contains status information. The following list shows this information, as it appears from left to right:

Status For example, **1B** is displayed while cicsterm is connecting to a server. Displayed at columns 1 – 3.

Terminal name
Also referred to as *LU Name*. Columns 4 – 7.

Action
For example, **X-System**, indicating that you cannot enter text in the terminal window because cicsterm is waiting for a response from the server. Columns 9 – 16.

Error number
Errors in the form CCLNNNN, relating to the CICS Transaction Gateway. Columns 17 – 24.

Server name
The server to which cicsterm is connected. Columns 27 – 35.

Upper case
An up arrow is displayed when the Shift key is pressed. Column 42.

Caps Lock
A capital A is displayed when Caps Lock is on. Column 43.

Insert on
The caret symbol (^) is displayed if text will be inserted, rather than overwriting existing text. If you have difficulty seeing the caret, change the font face and size, or use a screen magnifier to increase the size of the status line. Column 52.

Cursor position
The cursor position, in the form ROW/COLUMN, where ROW is a two-digit number, and COLUMN a three-digit number. The top left of the screen is 01/001. Column 75–80.

Note: You might need to change the default behavior of your screen reader if it reads only the last digit of the cursor position. Customize your screen reader to specify that columns 75–80 of the status row are to be treated as one field. This will cause the full area to be read when any digit changes.

The cicsterm -? command

After issuing the `cicsterm -?` command, use the up arrow key to move from the **OK** button to the list of messages. Use the up and down arrow keys to move through the messages. Press **Tab** and then **Enter** when done.

Glossary

This glossary defines special terms used in the CICS Transaction Gateway library.

3270 emulation
The use of software that enables a client to emulate an IBM 3270 display station or printer, and to use the functions of an IBM host system.

abnormal end of task (abend)
The termination of a task, job, or subsystem because of an error condition that recovery facilities cannot resolve.

Advanced program-to-program communication (APPC)
An implementation of the SNA/SDLC LU 6.2 protocol that allows interconnected systems to communicate and share the processing of programs. The Client daemon uses APPC to communicate with CICS server systems.

APAR See *Authorized program analysis report*.

API Application programming interface.

applet A small application program that performs a specific task and is usually portable between operating systems. Often written in Java, applets can be downloaded from the Internet and run in a Web browser.

application identifier
The name by which a CICS system is known in a network of interconnected CICS systems. CICS Transaction Gateway application identifiers do not need to be defined in SYS1.VTAMLST. The CICS APPLID is specified in the APPLID system initialization parameter.

application programming interface (API)
A functional interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program.

APPLID
See *application identifier*.

ARM See *automatic restart management*.

Authorized program analysis report (APAR)
A request for correction of a defect in a current release of an IBM-supplied program.

ATI See *automatic transaction initiation*.

attach In SNA, the request unit that flows on a session to initiate a conversation.

Attach Manager
The component of APPC that matches attaches received from remote computers to accepts issued by local programs.

autoinstall
A method of creating and installing resources dynamically as terminals log on, and deleting them at logoff.

automatic restart manager
A z/OS recovery function that can improve the availability of specific batch jobs or started tasks, and therefore result in faster resumption of productive work. Acronym: ARM.

automatic transaction initiation (ATI)
The initiation of a CICS transaction by an internally generated request, for example, the issue of an EXEC CICS START command or the reaching of a transient data trigger level. CICS resource definition can associate a trigger level and a transaction with a transient data destination. When the number of records written to the destination reaches the trigger level, the specified transaction is automatically initiated.

bean A definition or instance of a JavaBeans™ component. See also *JavaBeans*.

bean-managed transaction
A transaction where the J2EE bean itself is responsible for administering transaction tasks such as committal or rollback. See also *container-managed transaction*.

BIND command
In SNA, a request to activate a session between two logical units (LUs).

business logic
The part of a distributed application that is concerned with the application logic rather than the user interface of the application. Compare with *presentation logic*.

CA See *certificate authority*.

callback
A way for one thread to notify another application thread that an event has happened.

certificate authority
In computer security, an organization that issues certificates. The

certificate authority authenticates the certificate owner's identity and the services that the owner is authorized to use. It issues new certificates and revokes certificates from users who are no longer authorized to use them.

change-number-of-sessions (CNOS)
An internal transaction program that regulates the number of parallel sessions between the partner LUs with specific characteristics.

channel
A channel is a set of containers, grouped together to pass data to CICS. There is no limit to the number of containers that can be added to a channel, and the size of individual containers is limited only by the amount of storage that you have available.

CICS connectivity components
A generic reference to the Client daemon, EXCI, and the IPIC protocol.

CICS on System/390®
A generic reference to the products CICS Transaction Server for z/OS, CICS for MVS/ESA™, CICS Transaction Server for VSE/ESA™, and CICS/VSE®.

CICS TS
Abbreviation of CICS Transaction Server.

class In object-oriented programming, a model or template that can be instantiated to create objects with a common definition and therefore, common properties, operations, and behavior. An object is an instance of a class.

classpath
In the execution environment, an environment variable keyword that specifies the directories in which to look for class and resource files.

Client API
The Client API is the interface used by Client applications to invoke services in CICS using the Client daemon. See External Call Interface, External Presentation Interface, and External Security Interface.

Client application
The client application is a user application written in a supported programming language, other than Java, that uses the Client API.

Client daemon
The Client daemon, process cclclnt, exists only on UNIX, Windows, and Linux. It manages network connections to CICS servers. It processes ECI, EPI, and ESI requests, sending and receiving the appropriate flows from the CICS server to satisfy the application requests. It uses the CLIENT section of ctg.ini for its configuration.

client/server
Pertaining to the model of interaction in distributed data processing in which a program on one computer sends a request to a program on another computer and awaits a response. The requesting program is called a client; the answering program is called a server.

CNOS See *Change-Number-of-Sessions*.

code page
An assignment of hexadecimal identifiers (code points) to graphic characters. Within a given code page, a code point can have only one meaning.

color mapping file
A file that is used to customize the 3270 screen color attributes on client workstations.

commit phase
The second phase in a XA process. If all participants acknowledge that they are prepared to commit , the transaction manager issues the commit request. If any participant is not prepared to commit the transaction manager issues a back-out request to all participants.

communication area (COMMAREA)
A communication area that is used for passing data both between programs within a transaction and between transactions.

configuration file
A file that specifies the characteristics of a program, system device, server or network.

connection
In data communication, an association established between functional units for conveying information.

In Open Systems Interconnection architecture, an association established by a given layer between two or more entities of the next higher layer for the purpose of data transfer.

In TCP/IP, the path between two protocol application that provides reliable data stream delivery service.

In Internet, a connection extends from a TCP application on one system to a TCP application on another system.

container
A container is a named block of data designed for passing information between programs. A container is a ″named COMMAREA″ that is not limited to 32KB. Containers are grouped together in sets called channels.

container-managed transaction
A transaction where the EJB container is responsible for administration of tasks such as committal or rollback. See also *bean-managed transaction*.

control table
In CICS, a storage area used to describe or define the configuration or operation of the system.

conversation
A connection between two programs over a session that allows them to communicate with each other while processing a transaction.

conversation security
In APPC, a process that allows validation of a user ID or group ID and password before establishing a connection.

daemon
A program that runs unattended to perform continuous or periodic systemwide functions, such as network control. A daemon may be launched automatically, such as when the operating system is started, or manually.

data link control (DLC)
A set of rules used by nodes on a data link (such as an SDLC link or a token ring) to accomplish an orderly exchange of information.

DBCS See *double-byte character set*.

dependent logical unit
A logical unit that requires assistance from a system services control point (SSCP) to instantiate an LU-to-LU session.

deprecated
Pertaining to an entity, such as a programming element or feature, that is supported but no longer recommended, and that might become obsolete.

digital certificate
An electronic document used to identify an individual, server, company, or some other entity, and to associate a public key with the entity. A digital certificate is issued by a certificate authority and is digitally signed by that authority.

digital signature
Information that is encrypted with an entity's private key and is appended to a message to assure the recipient of the authenticity and integrity of the message. The digital signature proves that the message was signed by the entity that owns, or has access to, the private key or shared secret symmetric key.

distributed application

An application for which the component application programs are distributed between two or more interconnected processors.

distributed processing

The processing of different parts of the same application in different systems, on one or more processors.

distributed program link (DPL)

A link that enables an application program running on one CICS system to link to another application program running in another CICS system.

DLL See *dynamic link library*.

domain

In the Internet, a part of a naming hierarchy in which the domain name consists of a sequence of names (labels) separated by periods (dots).

domain name

In TCP/IP, a name of a host system in a network.

domain name server

In TCP/IP, a server program that supplies name-to-address translation by mapping domain names to internet addresses. Synonymous with name server.

dotted decimal notation

The syntactical representation for a 32-bit integer that consists of four 8-bit numbers written in base 10 with periods (dots) separating them. It is used to represent IP addresses.

double-byte character set (DBCS)

A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. Contrast with *single-byte character set*.

DPL See *distributed program link*.

dynamic link library (DLL)

A collection of runtime routines made available to applications as required.

EBCDIC

See *Extended binary-coded decimal interchange code*.

ECI See *external call interface*.

EJB See *Enterprise JavaBeans*.

emulation program
A program that allows a host system to communicate with a workstation in the same way as it would with the emulated terminal.

emulator
A program that causes a computer to act as a workstation attached to another system.

encryption
The process of transforming data into an unintelligible form in such a way that the original data can be obtained only by using a decryption process.

enterprise bean
A Java component that can be combined with other resources to create J2EE applications. There are three types of enterprise beans: entity beans, session beans, and message-driven beans.

Enterprise JavaBeans
A component architecture defined by Sun Microsystems for the development and deployment of object-oriented, distributed, enterprise-level applications (J2EE).

environment variable
A variable that specifies the operating environment for a process. For example, environment variables can describe the home directory, the command search path, the terminal in use, and the current time zone.

EPI See *external presentation interface*.

ESI See *external security interface*.

Ethernet
A local area network that allows multiple stations to access the transmission medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by using collision detection and transmission. Ethernet uses carrier sense multiple access with collision detection (CSMA/CD).

EXCI See *External CICS Interface*.

external call interface (ECI)
A facility that allows a non-CICS program to run a CICS program. Data is exchanged in a COMMAREA as for normal CICS interprogram communication.

Extended binary-coded decimal interchange code (EBCDIC)
A coded character set of 256 8-bit characters developed for the representation of textual data.

extended logical unit of work (extended LUW)
A logical unit of work that is extended across successive ECI requests to the same CICS server.

External CICS Interface (EXCI)
The EXCI is an MVS application programming interface provided by CICS Transaction Server for z/OS that enables a non-CICS program to call a CICS program and to pass and receive data using a COMMAREA or container. The CICS application program is invoked as if linked-to by another CICS application program.

external presentation interface (EPI)
A facility that allows a non-CICS program to appear to CICS as one or more standard 3270 terminals. 3270 data can be presented to the user by emulating a 3270 terminal or by using a graphical user interface.

external security interface (ESI)
A facility that enables client applications to verify and change passwords for user IDs on CICS servers.

firewall
A configuration of software that prevents unauthorized traffic between a trusted network and an untrusted network.

gateway
A device or program used to connect two systems or networks.

gateway classes
The Gateway Classes are the Java class library used by Java Client applications to invoke services in CICS.

Gateway daemon
The Gateway daemon is a long-running Java process used only in remote mode. The Gateway daemon listens for network requests from remote Java Client applications. It issues these requests to CICS using the CICS connectivity components. These are the Client daemon on UNIX, Windows, and Linux platforms, and EXCI or IPIC on z/OS. The Gateway daemon runs the protocol listener threads, the connection manager threads, and the worker threads. It uses the GATEWAY section of ctg.ini (and on z/OS the STDENV file or the ctgenvvar script) for its configuration.

Gateway group
A collection of Gateway daemon instances, that uses the services of a single `ctgmaster`. The group provides a TCP/IP load balancing capability for XA transactions.

gateway token
Gateway tokens are used in the statistical data API. A token represents a specific Gateway daemon, once a connection is established successfully.

global transaction
A recoverable unit of work performed by one or more resource managers in a distributed transaction processing environment and coordinated by an external transaction manager.

host A computer that is connected to a network (such as the Internet or an SNA network) and provides an access point to that network. The host can be any system; it does not have to be a mainframe.

host address
An IP address that is used to identify a host on a network.

host ID
In TCP/IP, that part of the Internet address that defines the host on the network. The length of the host ID depends on the type of network or network class (A, B, or C).

host name
In the Internet suite of protocols, the name given to a computer. Sometimes, host name is used to mean the fully qualified domain name; other times, it is used to mean the most specific subname of a fully qualified domain name. For example, if mycomputer.city.company.com is the fully qualified domain name, either of the following may be considered the host name: mycomputer.city.company.com, mycomputer.

hover help
Information that can be viewed by holding a mouse over an item such as an icon in the user interface.

HTTP See *Hypertext Transfer Protocol*.

HTTPS
See *Hypertext Transfer Protocol Secure*.

Hypertext Transfer Protocol
In the Internet suite of protocols, the protocol that is used to transfer and display hypertext and XML documents.

Hypertext Transfer Protocol Secure
A TCP/IP protocol that is used by World Wide Web servers and Web browsers to transfer and display hypermedia documents securely across the Internet.

ID data

An ID data structure holds an individual result from a statistical API function.

iKeyman

A tool for maintaining digital certificates for JSSE.

independent logical unit

A logical unit (LU) that can both send and receive a BIND, and which supports single, parallel, and multiple sessions. See *BIND*.

Internet Architecture Board

The technical body that oversees the development of the internet suite of protocols known as TCP/IP.

Internet Protocol (IP)

In TCP/IP, a protocol that routes data from its source to its destination in an Internet environment.

interoperability

The capability to communicate, execute programs, or transfer data among various functional units in a way that requires the user to have little or no knowledge of the unique characteristics of those units.

IP Internet Protocol.

IPIC See IP interconnectivity (IPIC).

IP address

A unique address for a device or logical unit on a network that uses the IP standard.

IP interconnectivity (IPIC)

The IPIC protocol enables Distributed Program Link (DPL) access from a non-CICS program to a CICS program over TCP/IP, using the External Call Interface (ECI). IPIC passes and receives data using COMMAREAs, or containers.

J2EE See *Java 2 Platform Enterprise Edition*

J2EE Connector architecture (JCA)

A standard architecture for connecting the J2EE platform to heterogeneous enterprise information systems (EIS).

Java An object-oriented programming language for portable interpretive code that supports interaction among remote objects.

Java 2 Platform Enterprise Edition (J2EE)

An environment for developing and deploying enterprise applications, defined by Sun Microsystems Inc. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that allow multitiered, Web-based applications to be developed.

JavaBeans
As defined for Java by Sun Microsystems, a portable, platform-independent, reusable component model.

Java Client application
The Java client application is a user application written in Java, including servlets and enterprise beans, that uses the Gateway classes.

Java Development Kit (JDK)
The name of the software development kit that Sun Microsystems provided for the Java platform, up to and including v 1.1.x. Sometimes used erroneously to mean the Java platform or as a generic term for any software developer kits for Java.

JavaGateway
The URL of the CICS Transaction Gateway with which the Java Client application will communicate. The JavaGateway takes the form `protocol://address:port`. These protocols are supported: `tcp://`, `ssl://`, and `local:`. The CICS Transaction Gateway runs with the default port value of 2006. This parameter is not relevant if you are using the protocol `local:`. For example, you might specify a JavaGateway of `tcp://ctg.business.com:2006`. If you specify the protocol as `local:` you will connect directly to the CICS server, bypassing any CICS Transaction Gateway servers.

Java Native Interface (JNI)
A programming interface that allows Java code running in a Java virtual machine to work with functions that are written in other programming languages.

Java Runtime Environment (JRE)
A subset of the Java Software Development Kit (SDK) that supports the execution, but not the development, of Java applications. The JRE comprises the Java Virtual Machine (JVM), the core classes, and supporting files.

Java Secure Socket Extension (JSSE)
A Java package that enables secure Internet communications. It implements a Java version of the Secure Sockets Layer (SSL) and Transport Layer Security (TSL) protocols and supports data encryption, server authentication, message integrity, and optionally client authentication.

Java virtual machine (JVM)
A software implementation of a processor that runs compiled Java code (applets and applications).

JDK See *Java development kit (JDK)*.

JCA See *J2EE Connector Architecture (JCA)*.

JNI See *Java Native Interface (JNI)*.

JRE See *Java Runtime Environment*

JSSE See *Java Secure Socket Extension (JSSE)*.

JVM See *Java Virtual Machine (JVM)*.

keyboard mapping
A list that establishes a correspondence between keys on the keyboard and characters displayed on a display screen, or action taken by a program, when that key is pressed.

key ring
In the JSSE protocol, a file that contains public keys, private keys, trusted roots, and certificates.

local mode
"Local mode" describes the use of the CICS Transaction Gateway *local* protocol. The Gateway daemon is not used in local mode.

local transaction
A recoverable unit of work managed by a resource manager and not coordinated by an external transaction manager

logical unit (LU)
In SNA, a port through which an end user accesses the SNA network in order to communicate with another end user and through which the end user accesses the functions provided by system services control points (SSCP). An LU can support at least two sessions, one with an SSCP and one with another LU, and may be capable of supporting many sessions with other logical units. See *network addressable unit*, *primary logical unit*, *secondary logical unit*.

logical unit 6.2 (LU 6.2)
A type of logical unit that supports general communications between programs in a distributed processing environment.

The LU type that supports sessions between two applications using APPC.

logical unit of work (LUW)
A recoverable unit of work performed within CICS.

LU-LU session
In SNA, a session between two logical units (LUs) in an SNA network. It provides communication between two end users, or between an end user and an LU services component.

LU-LU session type 6.2
In SNA, a type of session for communication between peer systems. Synonymous with APPC protocol.

LUW	See *logical unit of work*.
managed mode	Describes an environment in which connections are obtained from connection factories that the J2EE server has set up. Such connections are owned by the J2EE server.
medium access control (MAC) sublayer	One of two sublayers of the ISO Open Systems Interconnection data link layer proposed for local area networks by the IEEE Project 802 Committee on Local Area Networks and the European Computer Manufacturers Association (ECMA). It provides functions that depend on the topology of the network and uses services of the physical layer to provide services to the logical link control (LLC) sublayer. The OSI data link layer corresponds to the SNA data link control layer.
method	In object-oriented programming, an operation that an object can perform. An object can have many methods.
mode	In SNA, a set of parameters that defines the characteristics of a session between two LUs.
name server	In TCP/IP, synonym for Domain Name Server. In Internet communications, a host that translates symbolic names assigned to networks and hosts into Internet addresses.
network address	In SNA, an address, consisting of subarea and element fields, that identifies a link, link station, or network addressable unit (NAU). Subarea nodes use network addresses; peripheral nodes use local addresses. The boundary function in the subarea node to which a peripheral node is attached transforms local addresses to network addresses and vice versa. See also *network name*.
network addressable unit (NAU)	In SNA, a logical unit, a physical unit, or a system services control point. The NAU is the origin or the destination of information transmitted by the path control network. See also *logical unit, network address, network name*.
network name	In SNA, the symbolic identifier by which end users refer to a network addressable unit (NAU), link station, or link. See also *network address*.
node type	In SNA, a designation of a node according to the protocols it supports and the network addressable units (NAUs) it can contain. Four types

are defined: 1, 2, 4, and 5. Type 1 and type 2 nodes are peripheral nodes; type 4 and type 5 nodes are subarea nodes.

nonmanaged mode
An environment in which the application is responsible for generating and configuring connection factories. The J2EE server does not own or know about these connection factories and therefore provides no Quality of Service facilities.

object
In object-oriented programming, a concrete realization of a class that consists of data and the operations associated with that data.

object-oriented (OO)
Describing a computer system or programming language that supports objects.

one-phase commit
A protocol with a single commit phase, that is used for the coordination of changes to recoverable resources when a single resource manager is involved.

pacing
A technique by which a receiving station controls the rate of transmission of a sending station to prevent overrun.

parallel session
In SNA, two or more concurrently active sessions between the same two LUs using different pairs of network addresses. Each session can have independent session parameters.

PING
In Internet communications, a program used in TCP/IP networks to test the ability to reach destinations by sending the destinations an Internet Control Message Protocol (ICMP) echo request and waiting for a reply.

partner logical unit (PLU)
In SNA, the remote participant in a session.

partner transaction program
The transaction program engaged in an APPC conversation with a local transaction program.

PLU
See *primary logical unit* and *partner logical unit*.

port
An endpoint for communication between devices, generally referring to a logical connection. A 16-bit number identifying a particular Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) resource within a given TCP/IP node.

prepare phase
The first phase of a XA process in which all participants are requested to confirm readiness to commit.

presentation logic
The part of a distributed application that is concerned with the user interface of the application. Compare with *business logic*.

primary logical unit (PLU)
In SNA, the logical unit that contains the primary half-session for a particular logical unit-to-logical unit (LU-to-LU) session. See also *secondary logical unit*.

protocol boundary
The signals and rules governing interactions between two components within a node.

Query strings
Query strings are used in the statistical data API. A query string is an input parameter, specifying the statistical data to be retrieved.

Resource Access Control Facility (RACF)
An IBM licensed program that provides access control by identifying users to the system; verifying users of the system; authorizing access to protected resources; logging detected unauthorized attempts to enter the system; and logging detected accesses to protected resources.

region In workload management on CICS Transaction Gateway for Windows, an instance of a CICS server.

remote mode
"Remote mode" describes the use of one of the supported CICS Transaction Gateway network protocols to connect to the Gateway daemon.

remote procedure call (RPC)
A protocol that allows a program on a client computer to run a program on a server.

request unit (RU)
In SNA, a message unit that contains control information such as a request code, or function management (FM) headers, end-user data, or both.

request/response unit
A generic term for a request unit or a response unit. See also *request unit* and *response unit*.

response file
A file that contains predefined values that is used instead of someone having to enter those values one at a time. See *CID methodology*.

response unit (RU)
A message unit that acknowledges a request unit; it may contain prefix information received in a request unit.

resource group ID
A resource group ID is a logical grouping of resources, grouped for statistical purposes. A resource group ID is associated with a number of resource group statistics, each identified by a statistic ID.

resource ID
A resource ID refers to a specific resource. Information about the resource is included in resource-specific statistics. Each statistic is identified by a statistic ID.

resource manager
The participant in a transaction responsible for controlling access to recoverable resources. In terms of the CICS resource adapters this is represented by an instance of a ConnectionFactory.

Resource Recovery Services (RRS)
A z/OS facility that provides two-phase sync point support across participating resource managers.

Result set
A result set is a set of data calculated or recorded by a statistical API function.

Result set token
A result set token is a reference to the set of results returned by a statistical API function.

rollback
An operation in a transaction that reverses all the changes made during the unit of work. After the operation is complete, the unit of work is finished. Also known as a backout.

RU Request unit. Response unit.

RPC See *remote procedure call*.

SBCS See *single-byte character set*.

secondary logical unit (SLU)
In SNA, the logical unit (LU) that contains the secondary half-session for a particular LU-LU session. Contrast with primary logical unit. See also *logical unit*.

Secure Sockets Layer (SSL)
A security protocol that provides communication privacy. SSL enables client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery. SSL applies only to internet protocols, and is not applicable to SNA.

servlet
A Java program that runs on a Web server and extends the server's

functionality by generating dynamic content in response to Web client requests. Servlets are commonly used to connect databases to the Web.

session limit
In SNA, the maximum number of concurrently active logical unit to logical unit (LU-to-LU) sessions that a particular logical unit (LU) can support.

single-byte character set (SBCS)
A character set in which each character is represented by 1 byte. Contrast with double-byte character set.

sign-on capable terminal
A sign-on capable terminal allows sign-on transactions, either CICS-supplied (CESN) or user-written, to be run. Contrast with sign-on incapable terminal.

SIT See *system initialization table*.

SNA sense data
An SNA-defined encoding of error information In SNA, the data sent with a negative response, indicating the reason for the response.

SNASVCMG mode name
The SNA service manager mode name. This is the architecturally-defined mode name identifying sessions on which CNOS is exchanged. Most APPC-providing products predefine SNASVCMG sessions.

socket A network communication concept, typically representing a point of connection between a client and a server. A TCP/IP socket will normally combine a host name or IP address, and a port number.

SSL See *Secure Sockets Layer (SSL)*.

SSLight
An implementation of SSL, written in Java, and no longer supported by CICS Transaction Gateway.

statistic data
A statistic data structure holds individual statistical result returned after calling a statistical API function.

statistic group
A statistic group is a generic term for a collection of statistic IDs.

statistic ID
A statistic ID is a label refering to a specific statistic. A statistic ID is used to retrieve specific statistical data, and always has a direct relationship with a statistic group.

system initialization table
A table containing parameters used to start a CICS control region.

System Management Interface Tool (SMIT)
An interface tool of the AIX operating system for installing, maintaining, configuring, and diagnosing tasks.

standard error
In many workstation-based operating systems, the output stream to which error messages or diagnostic messages are sent.

subnet
An interconnected, but independent segment of a network that is identified by its Internet Protocol (IP) address.

subnet address
In Internet communications, an extension to the basic IP addressing scheme where a portion of the host address is interpreted as the local network address.

sync point
A logical point in the execution of program where the changes made by the program are consistent and complete, and can be committed. The output, which has been held up to that point, is sent to its destination, the input is removed from the message queues, and updates are made available to other applications. When a program terminates abnormally, CICS recovery and restart facilities do not backout updates prior to the last completed sync point.

Systems Network Architecture (SNA)
An architecture that describes the logical structure, formats, protocols, and operational sequences for transmitting information units through the networks and also the operational sequences for controlling the configuration and operation of networks.

System SSL
An implementation of SSL, no longer supported by CICS Transaction Gateway on z/OS.

TCP62 SNA logical unit type 62 (LU62) protocol encapsulated in TCP/IP. This allows APPC applications to communicate over a TCP/IP Network without changes to the applications.

TCP/IP
See *Transmission Control Protocol/Internet Protocol*.

TCP/IP load balancing
The ability to distribute TCP/IP connections across target servers.

terminal emulation
The capability of a microcomputer or personal computer to operate as

if it were a particular type of terminal linked to a processing unit and to access data. See also *emulator, emulation program*.

thread A stream of computer instructions that is in control of a process. In some operating systems, a thread is the smallest unit of operation in a process. Several threads can run concurrently, performing different jobs.

timeout
A time interval that is allotted for an event to occur or complete before operation is interrupted.

TLS See *Transport Layer Security (TLS)*.

token-ring network
A local area network that connects devices in a ring topology and allows unidirectional data transmission between devices by a token-passing procedure. A device must receive a token before it can transmit data.

trace A record of the processing of a computer program. It exhibits the sequences in which the instructions were processed.

transaction manager
A software unit that coordinates the activities of resource managers by managing global transactions and coordinating the decision to commit them or roll them back.

transaction program
A program that uses the Advanced Program-to-Program Communications (APPC) application programming interface (API) to communicate with a partner application program on a remote system.

Transmission Control Protocol/Internet Protocol (TCP/IP)
An industry-standard, nonproprietary set of communications protocols that provide reliable end-to-end connections between applications over interconnected networks of different types.

Transport Layer Security (TLS)
A security protocol that provides communication privacy. TLS enables client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery. TLS applies only to internet protocols, and is not applicable to SNA. TLS is also known as SSL 3.1.

two-phase commit
A protocol with both a prepare and a commit phase, that is used for the coordination of changes to recoverable resources when more than one resource manager is used by a single transaction.

type 2.0 node
A node that attaches to a subarea network as a peripheral node and provides a range of end-user services but no intermediate routing services.

type 2.1 node
An SNA node that can be configured as an endpoint or intermediate routing node in a network, or as a peripheral node attached to a subarea network.

Uniform Resource Locator (URL)
A sequence of characters that represent information resources on a computer or in a network such as the Internet. This sequence of characters includes (a) the abbreviated name of the protocol used to access the information resource and (b) the information used by the protocol to locate the information resource.

unit of recovery (UR)
A defined package of work to be performed by the RRS.

unit of work (UOW)
A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or at a user-requested sync point. It ends either at a user-requested sync point or at the end of a transaction.

user session
Any APPC session other than a SNASVCMG session.

verb A reserved word that expresses an action to be taken by an application programming interface (API), a compiler, or an object program.

In SNA, the general name for a transaction program's request for communication services.

version string
A character string containing version information about the statistical data API.

Web browser
A software program that sends requests to a Web server and displays the information that the server returns.

Web server
A software program that responds to information requests generated by Web browsers.

wide area network (WAN)
A network that provides communication services to a geographic area

larger than that served by a local area network or a metropolitan area network, and that may use or provide public communication facilities.

wrapping trace

A configuration in which the **Maximum Client wrap size** setting is greater than 0. The total size of Client daemon binary trace files is limited to the value specified in the **Maximum Client wrap size** setting. With standard I/O tracing, two files, called `cicscli.bin` and `cicscli.wrp`, are used; each can be up to half the size of the **Maximum Client wrap size**.

XA requests

An XA request is any request sent or received by the CICS Transaction Gateway in support of an XA transaction. These requests include the XA commands commit, complete, end, forget, prepare, recover, rollback, and start.

XA transaction

A global transaction that adheres to the X/Open standard for distributed transaction processing (DTP.)

Index

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply in the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM United Kingdom Laboratories, MP151, Hursley Park, Winchester, Hampshire, England, SO21 2JN. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX
AnyNet
AS/400
CICS
CICS/400
CICS/ESA
CICS/VSE
DB2
Domino
Hummingbird
IBM
IBM
IBMLink
IMS
iSeries
MQSeries
MVS
MVS/ESA
Notes
OS/2
OS/390
POWER
pSeries
RACF
Redbooks
RETAIN
RMF
RS/6000
SAA
SP2
System/390
Tivoli
TXSeries
VisualAge
VSE/ESA
VTAM
WebSphere
z/OS
zSeries

Microsoft, Windows, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Intel, Intel Inside® (logos), MMX and Pentium® are trademarks of Intel Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Sending your comments to IBM

If you especially like or dislike anything about this book, use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Limit your comments to the information in this book and the way in which the information is presented.

To ask questions, make comments about the functions of IBM products or systems, or to request additional publications, contact your IBM representative or your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - +44 1962 842327 (if you are outside the UK)
  - 01962 842327 (if you are in the UK)
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Program Number: 5724-I81, 5655-R25 and 5724-J09

SC34-6758-02

Spine information:

IBM CICS Transaction Gateway **Programming Guide** Version 7.1